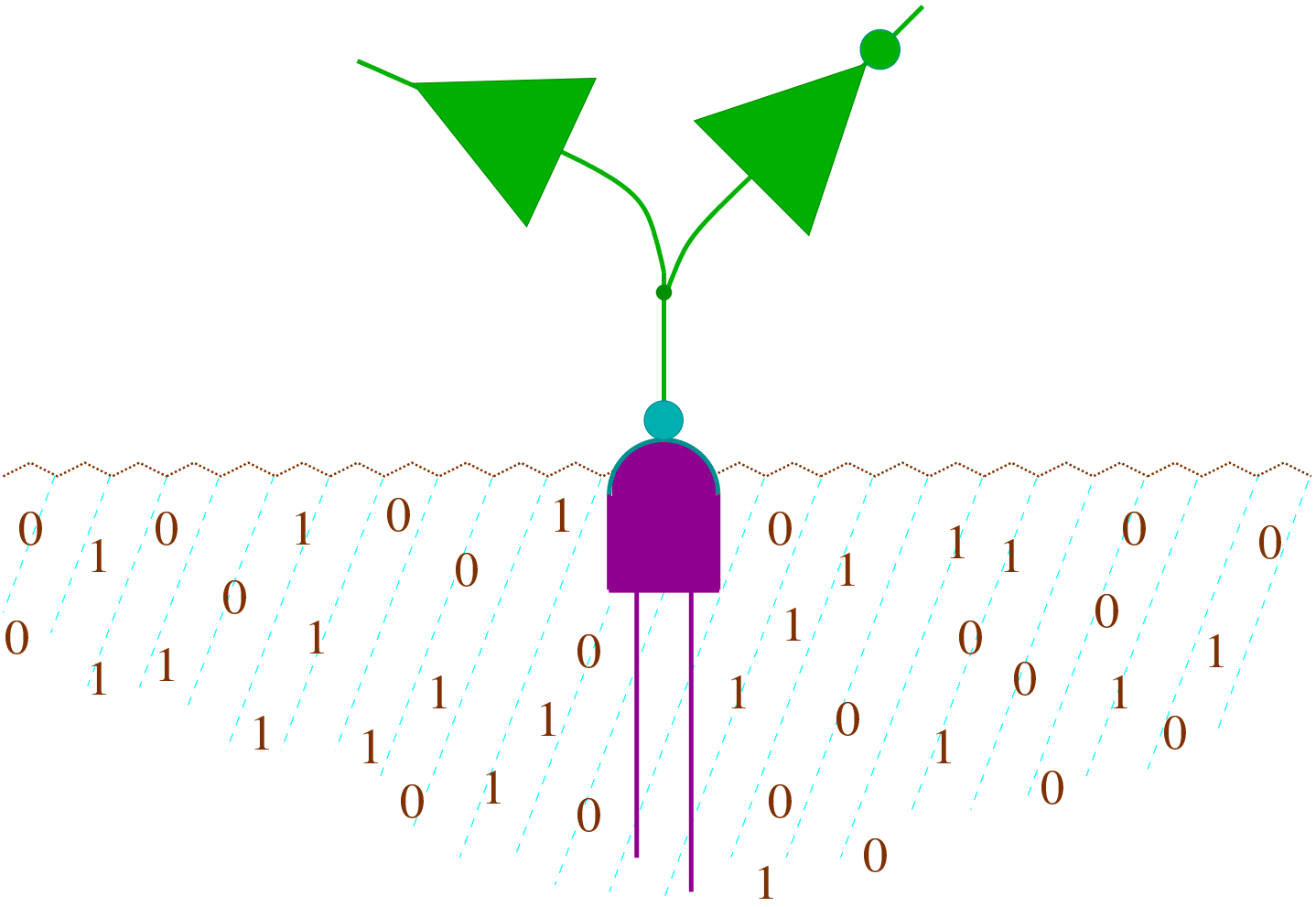
“from the ground up”

logic gates et al.

Daniele Giacomini appunti2@gmail.com

draft-2013.03



GNU GPL GNU FDL CC BY–SA

“**from the ground up**” -- Copyright © 2013 Daniele Giacomini

Via Morganella Est, 21 -- I-31050 Ponzano Veneto (TV) -- appunti2@gmail.com

You can redistribute this work and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version, with the following exceptions and clarifications:

- Quotations and samples of other works are not subject to the scope of the license of this work.
- If you modify this work and/or reuse it partially, under the terms of the license: it is your responsibility to avoid misrepresentation of opinion, thought and/or feeling of other than you; the notices about changes and the references about the original work, must be kept and evidenced conforming to the new work characteristics; you may add or remove quotations and/or samples of other works; you are required to use a different name for the new work.

Permission is also granted to copy, distribute and/or modify this work under the terms of the GNU Free Documentation License (FDL), either version 1.3 of the License, or (at your option) any later version published by the Free Software Foundation (FSF); with no Invariant Sections, with no Front-Cover Text, and with no Back-Cover Texts.

Permission is also granted to copy, distribute and/or modify this work under the terms of the Creative Commons Attribution-ShareAlike License, version 3.0, as published by Creative Commons at *http://creativecommons.org/licenses/by-sa/3.0/* .

This work is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Table of contents

Foreword IV

Part i Logic circuits 1

Traditional logic integrated circuits 243

References 301

Part ii Simple CPU 303

References 981

Index i

Foreword

«

Dear reader, this work is almost a collection of notes on the fields I like to study. I do it using English because this is the language for the information technology when the concepts are beyond the common interest and because I would like to practice with English as well. As you can understand from these few lines, I might be a good English reader, but surely I am not a good writer. If you are interested on the topic of this book and you will read it, please be prepared and please forgive me, but if you are connected and you like it to do, I would appreciate if you send me a note about the phrases that I might correct (it is enough that you write something like: “you wrote this... but you should write that...”). My email address is appunti2@gmail.com.

Who am I? I was a technical teacher at high school. I was excited in 1981 when I could use a Burroughs B91: I learned to write COBOL programs and experienced with other Burroughs proprietary languages for the system maintenance. Then, I felt excited again in 1995 when I started studying GNU/Linux systems and I dreamed (again) a world of free software, free knowledge (and peaceful living); I also wrote a free book about GNU/Linux and what it can be done with it (of course, I used my mother language for it). But I found that things do not work as simple as in my dream —especially with my teaching experience— because in human nature the ‘freedom’ is something that is not appreciated if it is ‘free’. Freedom is not just a matter of money or laws, but when it should be applied to the information technology the freedom requires knowledge: information technology is something on what we all depend for our life, but depending on what we can not verify —because we do not understand in-depth or because, for unknown reasons, we prefer what is not open source— we will be just slave. Anyway, although my thoughts are so catastrophic, I am still attracted from what I felt in my young ages, when I imagined that computers could be built simply with discrete logic (and all the software could be written from scratch)... :-)

Daniele Giacomini
Via Morganella Est, 21
I-31050 Ponzano Veneto (TV)
Italy
appunti2@gmail.com
http://it.linkedin.com/pub/daniele-giacomini/4b/364/6b2
http://informaticalibera.net

Part i

Logic circuits

1 Logic functions and logic gates 5

1.1 NOT 7
1.2 'not-inverter' or 'buffer' 7
1.3 AND 8
1.4 OR 9
1.5 XOR 9
1.6 Logic networks 10
1.7 Operator precedence 13
1.8 Irrelevant values inside the truth table 13
1.9 Equivalence 13
1.10 Sum of produtcs 17
1.11 Karnaugh maps 19
1.12 Logic gates and multiple lines 24
1.13 Electronic logic gates 25
1.14 HDL 28

2 Introduction to Verilog 31

2.1 Minimal notions 33
2.2 Hello world! 33
2.3 Compiler directives 38
2.4 Signal values 41
2.5 Literal constants 41
2.6 Data types 43
2.7 Ports 46
2.8 Parameters 48
2.9 Module declarations 48
2.10 Module instance 51

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

2.11 Fully qualified path names 51
2.12 Primitives 52
2.13 Expressions 54
2.14 Vector and array addressing 58
2.15 Strings 59
2.16 Functions 60
2.17 Netlist modules: combinational circuits 62
2.18 Behavioral modules 68
2.19 Procedural assignments 69
2.20 Blocking delays 71
2.21 Wait for a level event 72
2.22 Wait for an event expressions 74
2.23 Event variables 76
2.24 System tasks and functions 76
2.25 Control structures 78
2.26 Thread control 81

3 Combinational circuits 83

3.1 Decoder 87
3.2 Demultiplexer 90
3.3 Multiplexer 93
3.4 Multiplexer and demultiplexer with paralel I/O 96
3.5 Binary encoder 99
3.6 Priority encoder 102
3.7 Logic units 106
3.8 Shift and rotation 107
3.9 Addition 114
3.10 Subtraction 119
3.11 Addition and subtraction together 123
3.12 Carry lookahead 125
3.13 2's complement 132

3.14 Multiplication . . . 133
3.15 Division . . . 142
3.16 Magnitude comparator . . . 145
4 Arithmetic-logic unit . . . 149
4.1 Status flags . . . 150
4.2 Size truncation . . . 152
4.3 Sign inversion . . . 154
4.4 Addition and subtraction . . . 156
4.5 Shift and rotation . . . 158
4.6 Magnitude comparation . . . 162
4.7 Multiplication . . . 165
4.8 Division . . . 171
4.9 Logic unit . . . 176
4.10 Joining all the module together . . . 177
4.11 Bit sliced ALU . . . 179
5 Latch and flip-flop . . . 185
5.1 Propagation delay . . . 185
5.2 Truth tables . . . 187
5.3 SR latch . . . 188
5.4 Bounce-free switch . . . 193
5.5 Gated SR latches . . . 193
5.6 SR flip-flop . . . 195
5.7 Time: setup/hold and recovery/removal . . . 198
5.8 D latch and D flip-flop . . . 199
5.9 T flip-flop . . . 203
5.10 JK flip-flop . . . 204
5.11 Troublesome JK flip-flops . . . 207
6 Registers . . . 211
6.1 Simple data register . . . 212

6.2 Shift register 216
6.3 Asynchronous counter with T flip-flop 217
6.4 Synchronous T flip-flop counters 219
6.5 Synchronous counters with D flip-flops 220
6.6 Synchronous counters with paralel load 224

7 Bus and control unit 229

7.1 Tri-state buffer 229
7.2 Hardcoded bus control unit 232
7.3 Microcode 236

Traditional logic integrated circuits 243

Data distributors or demultiplexers 244
Data selectors or multiplexers 257
Special circuits 271
Arithmetic units 275

References 301

Chapter 1

Logic functions and logic gates

1.1 NOT 7
1.2 ‘not-inverter’ or ‘buffer’ 7
1.3 AND 8
1.4 OR 9
1.5 XOR 9
1.6 Logic networks 10
1.7 Operator precedence 13
1.8 Irrelevant values inside the truth table 13
1.9 Equivalence 13
1.10 Sum of produtcs 17
1.11 Karnaugh maps 19
1.12 Logic gates and multiple lines 24
1.13 Electronic logic gates 25
1.14 HDL 28

active high 10 active low 10 AND 8 assert 10 boolean algebra 13 buffer 7 De Morgan’s laws 13 discrete logic 25 floating 25 fundamental product 17 hardware description language 28 HDL 28 high impedance 25 IC 25 integrated circuit 25 inverter 7 Karnaugh map 19 logic circuit 10 logic connective 5 logic gate 5 logic network 10 logic operator 5 logic variable 5 negate 10 NOT 7 not-inverter 7 OR 9 propagation delay 25 signal 10 sum of products 17 tri-state buffer 25 Verilog 28 XOR 9 Z 25

The term ***logic*** comes from the philosophy and it is the way by which something can be found ***true*** or ***false***. The concepts of ‘true’ and ‘false’ can be considered as the two elementary values of a ***logic variable***. There are different ways to represent the values of a logic variable and every context might have its own symbology, but when using

“from the ground up” draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

numbers —unless otherwise specified— it is custom to associate 1 to ‘true’ and 0 to ‘false’.

A ***logic expression***, or a ***logic function***, is the expression or the function that produces a true or false result. The components of a logic expression or function are listed below:

- declarative propositions – propositions that can be found either true or false;
- logic constants – either true or false;
- logic variables – variables that can represent only the values true and false;
- logic functions and operators – the functions or the operators used to connect the other logic components into a more complex function or expression.

Logical functions can be represented in different ways, depending on the context and on possible typographical limitations. Therefore, every time it is necessary to understand which is the adopted symbology, taking into account that the representation might change inside the same document; for example, the body text explaining a problem might use one symbology, whereas the pictures might have a different and more appropriate one. One particular way to represent a logic function consists on drawing little boxes, with one or more inputs and one single output: these are known as ***logic gates***.

Table 1.1. Some alternative notations regarding the common logic functions, associating the symbology used for the corresponding logic gates.

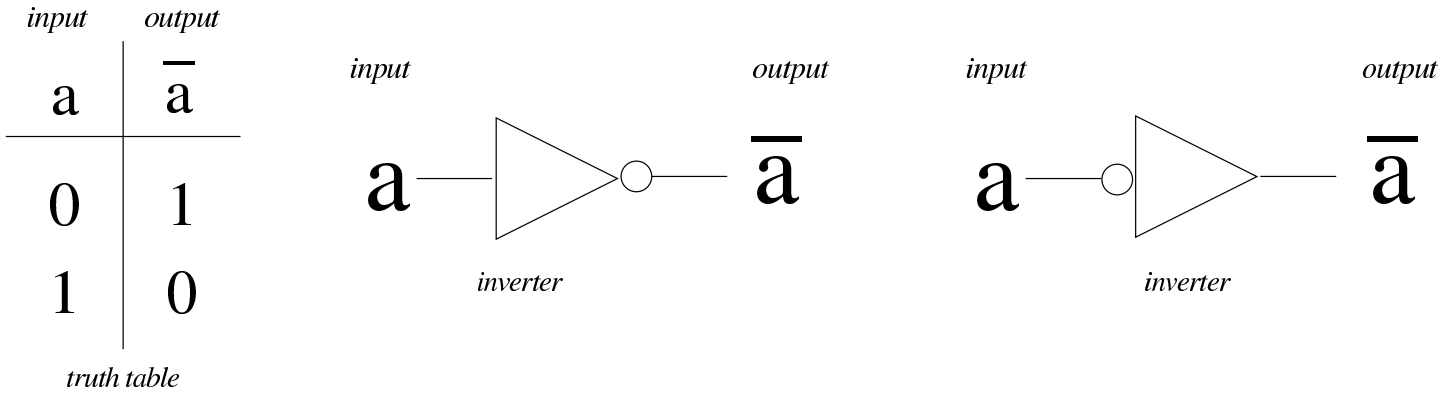
a	a				a — a
a	NOT a	a’	¬a	$\overline{a}$	a — $\overline{a}$
a b	a AND b	a ∧ b	a·b	ab	a, b — a·b
a b	a OR b	a ∨ b	a + b		a, b — a+b
a b	a XOR b	a ⊕ b			a, b — a⊕b

The ***truth tables*** are used to define the meaning of logic operators: all the values of the input variables are matched with the corresponding logic results.

1.1 NOT

The NOT function has a single input and produces the logic inversion. That is: if the input is true (1), the output is false (0) and vice versa. When the NOT function is represented in the form of a logic gate, it is known with the name ***inverter***.

Figure 1.2. NOT (inverter).

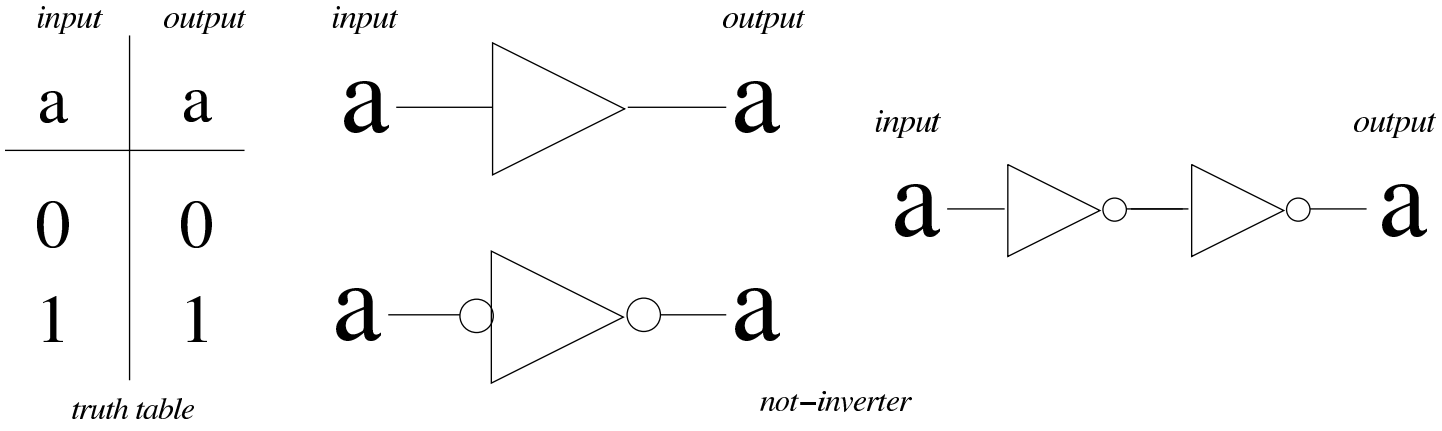


The figure above shows that there are two alternative way to represent an inverter gate: the first one is the most common.

1.2 `not-inverter' or `buffer'

Two consecutive inversions give just the original logic value; that is, "NOT (NOT A)" is just the same as "A". There is a logic gate that is known as ***not-inverter*** or ***buffer***: it does nothing from the logic point of view, but it has some meaning for other problems.

Figure 1.3. Buffer (not-inverter).



From the example shown for the inverter and the buffer logic gates, it should be noticed the use of the little bubble that means logic inversion. This bubble can appear at the input or at the output and it might be used also with other logic gates.

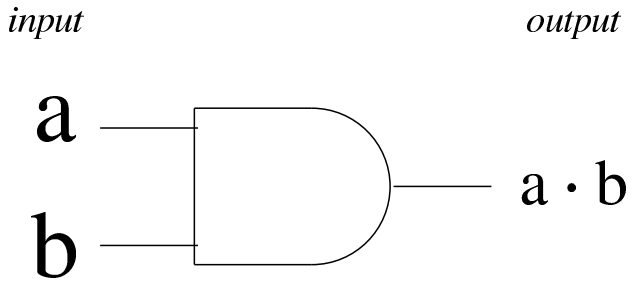
1.3 AND

The AND function takes two inputs and the result is true only if both inputs are true, otherwise the result is false.

Figure 1.4. AND.

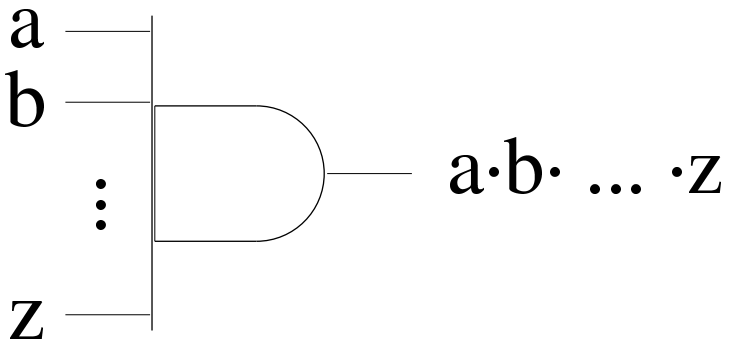
input		*output*
a	b	a · b
0	0	0
0	1	0
1	0	0
1	1	1

truth table



The AND logic function might have more than two input variables and the result is true only if all the inputs are also true. The same way, the AND logic gate might have more than two input lines, as it is shown in the following figure.

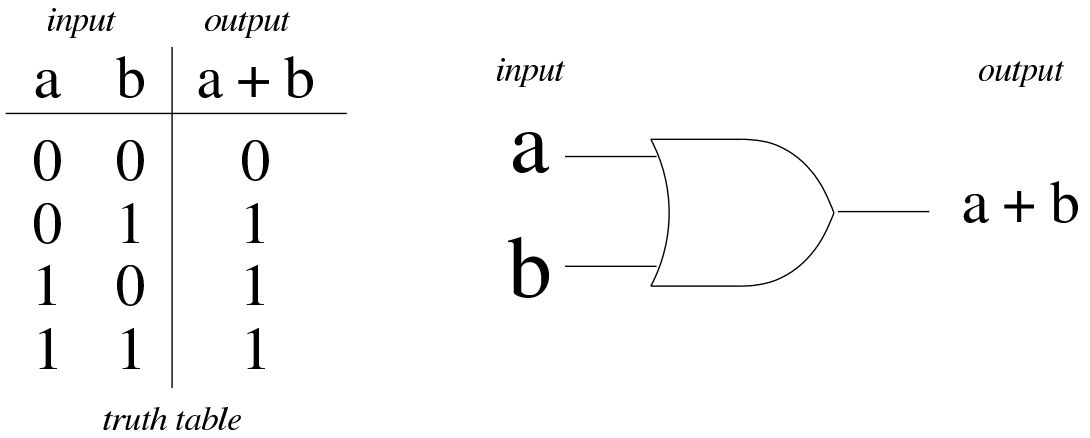
Figure 1.5. AND gate with more than two inputs.



1.4 OR

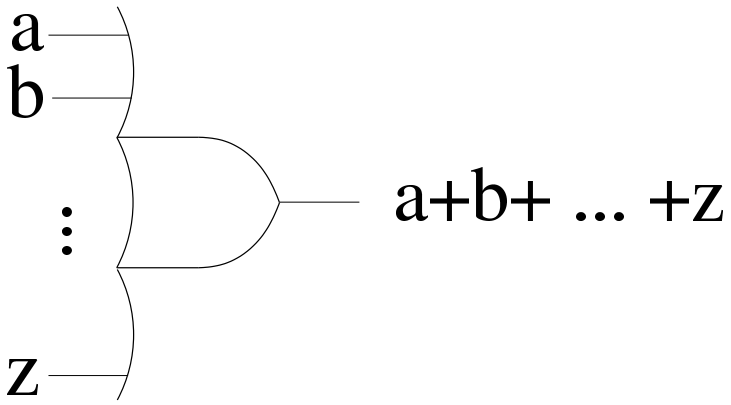
The OR logic function takes two inputs and the result is true if at least one input is true, otherwise the result is false.

Figure 1.6. OR.



The OR logic function might have more than two input variables and the result is true if at least one input is also true. The same way, the OR logic gate might have more than two input lines, as it is shown in the following figure.

Figure 1.7. OR gate with more than two inputs.



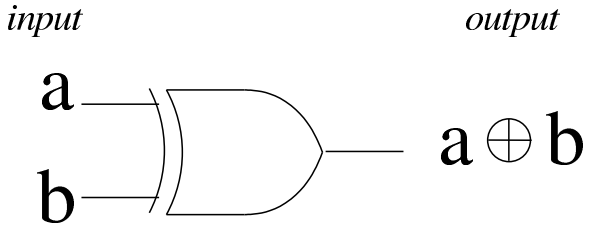
1.5 XOR

The XOR logic function (*exclusive or*) is derived from the other one: it takes two inputs and the result is true if only one input is true, otherwise the result is false.

Figure 1.8. XOR.

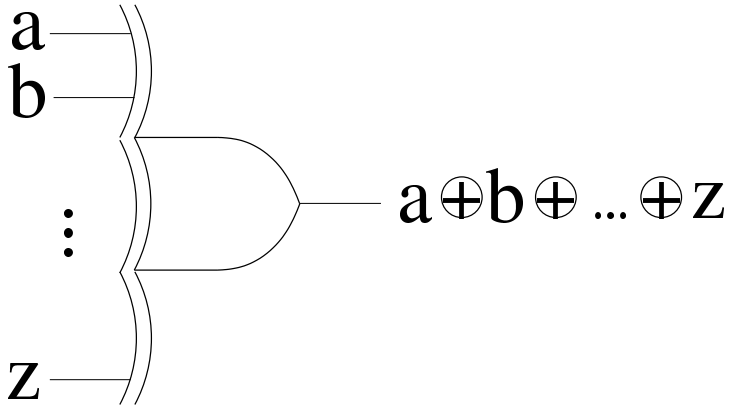
input		*output*
a	b	a⊕b
0	0	0
0	1	1
1	0	1
1	1	0

truth table



Even the XOR logic function might have more than two input variables and the result is true if only one input is also true. The same way, the XOR logic gate might have more than two input lines, as it is shown in the following figure.

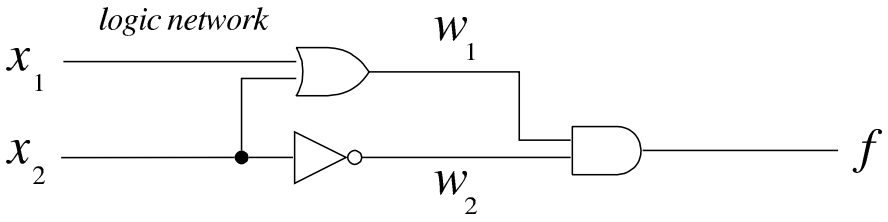
Figure 1.9. XOR gate with more than two inputs.



1.6 Logic networks

Elementary logic functions can be used together to build more complex ones. When using logic gates, connecting two or more gates results into a ***logic network*** or ***logic circuit***, which are equivalent definitions of the same thing.

Figure 1.10. A simple logic network.



logic function

$$f = (x_1 + x_2) \cdot \overline{x_2}$$

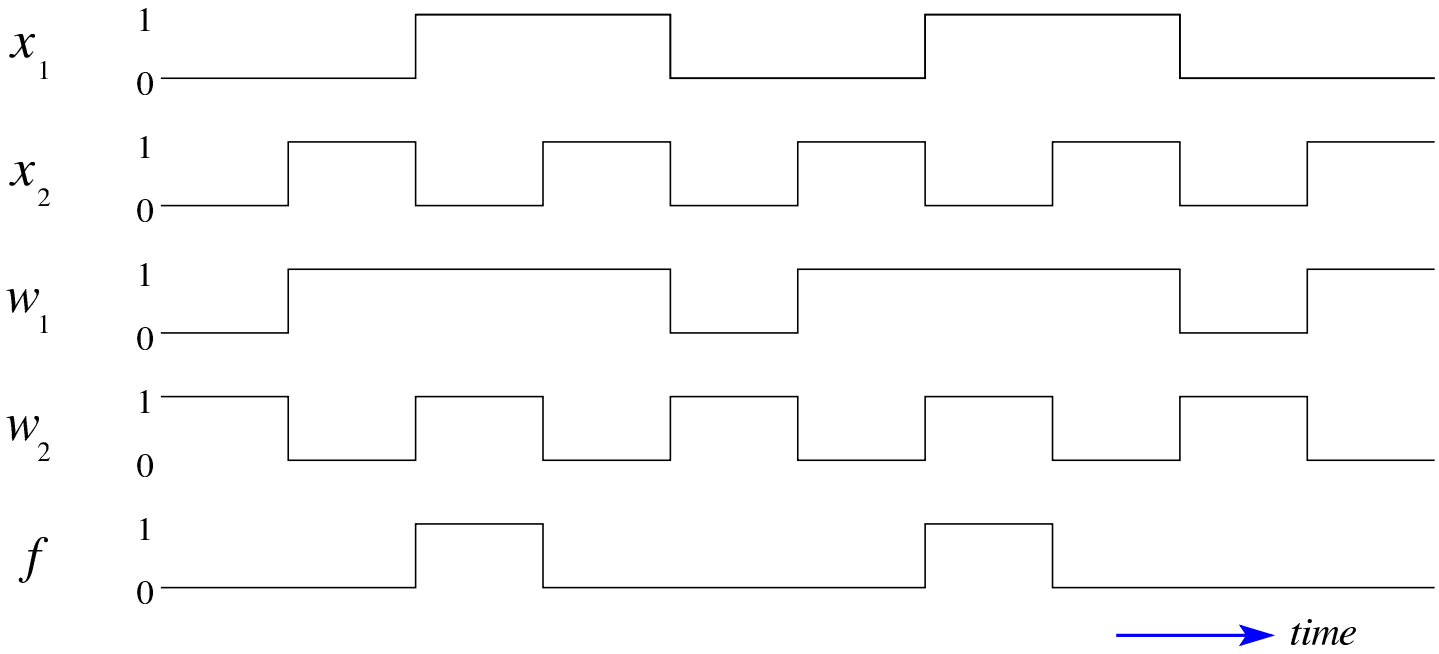
x_1	x_2	$f(x_1, x_2)$
0	0	0
0	1	0
1	0	1
1	1	0

truth table

The logic gates are connected together with wires; the figure above shows a name for every wire, where $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are the inputs of the overall logic function and $\boldsymbol{f}$ is the output. Each wire can carry a value of 0 or 1 that in electronics is represented by a tension (measured in Volts and known also as voltage). In electronic circuits there is usually a common connection known as ***ground***, because it carries the 0-volt point of reference for the other electric tensions. In a logic circuit, when a wire has the same voltage of the ground, it represents the value 0; when the electric tension is higher (or lower, depending on the specific technology used) the value that a wire represents is 1.

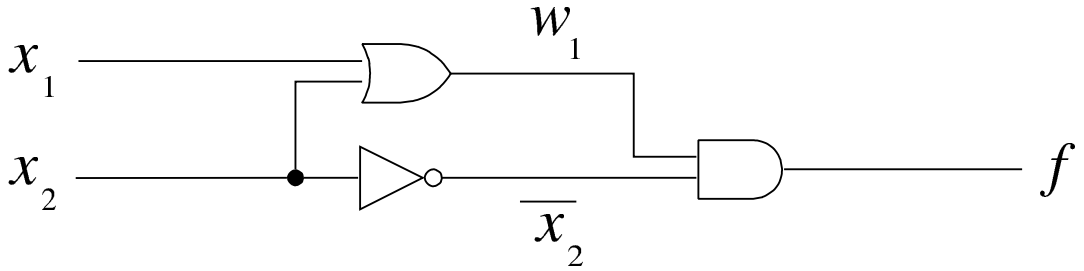
Together with the truth table, a logic network might have also a timing diagram, where the electric signals (the tensions) are shown dynamically, like the example below: when the line of the diagram is low it represents the value 0; when the line is high it represents the value 1.

Figure 1.11. Timing diagram.



The electric signal (tension) used to represent the value 1 might be positive or negative in relation to the ground potential. The choice to be positive or negative depends on the technology used to build the logic gates. When a timing diagram is drawn, the value of the signal used to represent 1 is always shown high, regardless if it is a positive or negative tension.

Figure 1.12. The simple logic network with a negated wire name.



When a wire is named, this name might be negated. The figure above shows the same simple logic network, where the previous wire $\boldsymbol{w}_2$ is replaced by the name $\overline{\boldsymbol{x}}_2$. In fact, the wire $\overline{\boldsymbol{x}}_2$ is obtained inverting the value at the wire $\boldsymbol{x}_2$ and so it is easier to read the logic network.

It is common to use the word ‘active’ to mean 1 on a wire, but when a wire is labeled with a negated name, the meaning of ‘active’ becomes ambiguous. The name of the wire implies the name of the signal it carries: a plain name (without negation) means that this is the name of the high signal or the name of the value 1 (active high); on

the other hand, a negated name is the name of a low signal or the name of a value 0 (active low). To express the meaning that the signal is the one that it is expected to be, based on the name of the wire, it is custom to use the words ***assert*** and ***negate***. For example, in the above figure, the wire $\boldsymbol{w}_1$ is asserted when it is high —or one— and negated when it is low —or zero—, whereas the wire $\overline{\boldsymbol{x}}_2$ is asserted when it is low —or zero— and it is negated when it is high —or one—.

1.7 Operator precedence

When the logic functions are written in the form of expressions, it is important to define the order of precedence, to avoid an excessive use of parentheses. This order is: NOT, AND, OR. For example, $\boldsymbol{a}+\boldsymbol{b}\cdot\overline{\boldsymbol{c}}$ means exactly $\boldsymbol{a}+(\boldsymbol{b}\cdot(\overline{\boldsymbol{c}}))$. It should be noticed that it works like the usual arithmetic, where multiplication has the precedence on the addition.

The precedence problem is not related to logic networks, because the connections define the order of evaluation.

It is not specified the precedence order for the XOR operator, because it is a derived function of the other ones. So, if ambiguity might occur with a logic expression, parentheses should be used.

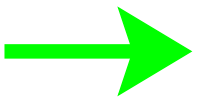
1.8 Irrelevant values inside the truth table

There are situation where truth tables of logic functions can be reduced, because for some conditions the value of some inputs are irrelevant. When an input value might be anything, it is used a letter 'X'. For example, the simple function shown on a previous section is $(\boldsymbol{x}_1+\boldsymbol{x}_2)\cdot\overline{\boldsymbol{x}}_2$ and it happens that if $\boldsymbol{x}_1$ is 0, it does not matter what value has $\boldsymbol{x}_2$, because the output remains 0.

Figure 1.13. Truth table simplified.

$f = (x_1 + x_2) \cdot \overline{x_2}$

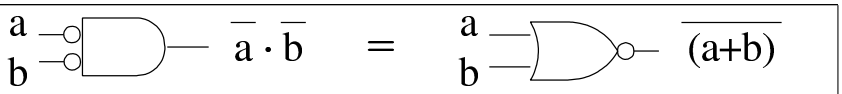
x_1	x_2	$f(x_1, x_2)$
0	0	0
0	1	0
1	0	1
1	1	0

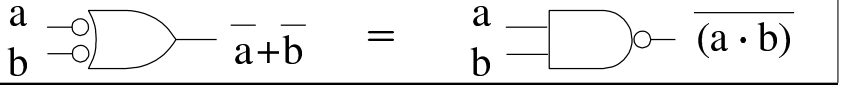


x_1	x_2	$f(x_1, x_2)$
0	X	0
1	0	1
1	1	0

1.9 Equivalence

Through the elementary logic operators it is possible to build complex functions, where there are different alternatives to obtain the same result. For example, the XOR function might be obtained as $(\overline{a}+b)\cdot(a+\overline{b})$ or as $(a+b)\cdot\overline{(a\cdot b)}$ or as $(a+b)\cdot(\overline{a}+\overline{b})$, or even more complex and less useful ways. To transform a logic function into another, producing the same result, come to help the De Morgan's laws:

$\overline{a}\cdot\overline{b}=\overline{(a+b)}$ 

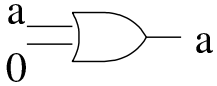
$\overline{a}+\overline{b}=\overline{(a\cdot b)}$ 

Putting together the truth tables of the basic logic functions with the De Morgan's laws, it is possible to find the equivalences shown in the following table; equivalences on which is based all the 'logic' algebra, better known as the boolean algebra (or Boole's algebra).

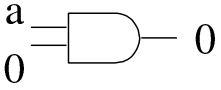
Table 1.15. Boolean algebra equivalences.

(1)	$a+0=a$	(2)	$a\cdot 0=0$
(3)	$a+1=1$	(4)	$a\cdot 1=a$
(5)	$a+a=a$	(6)	$a\cdot a=a$
(7)	$a+\overline{a}=1$	(8)	$a\cdot\overline{a}=0$
(9)	$a+b=b+a$	(10)	$a\cdot b=b\cdot a$
(11)	$a+(b+c)=(a+b)+c$	(12)	$a\cdot(b\cdot c)=(a\cdot b)\cdot c$
(13)	$a\cdot(b+c)=a\cdot b+a\cdot c$	(14)	$a+b\cdot c=(a+b)\cdot(a+c)$
(15)	$\overline{(a+b)}=\overline{a}\cdot\overline{b}$	(16)	$\overline{(a\cdot b)}=\overline{a}+\overline{b}$
(17)	$\overline{(\overline{a})}=a$		

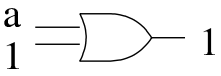
1. $a+0=a$



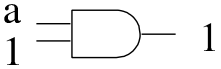
2. $a \cdot 0 = 0$



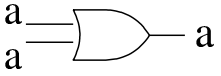
3. $a+1 = 1$



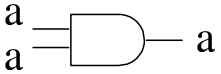
4. $a \cdot 1 = a$



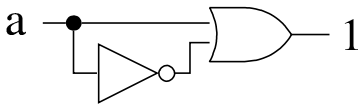
5. $a+a = a$

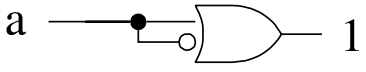


6. $a \cdot a = a$

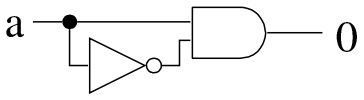


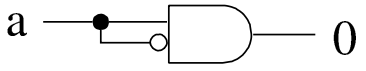
7. $a+\overline{a} = 1$



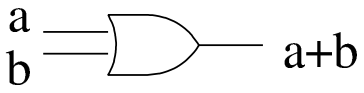


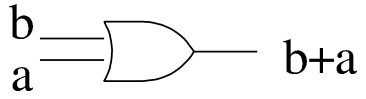
8. $a \cdot \overline{a} = 0$



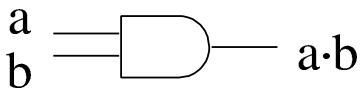


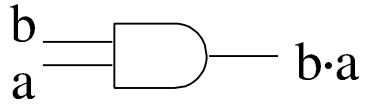
9. $a+b = b+a$



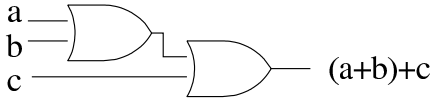


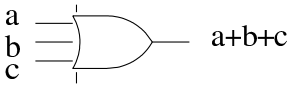
10. $a \cdot b = b \cdot a$



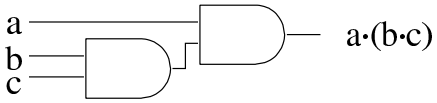


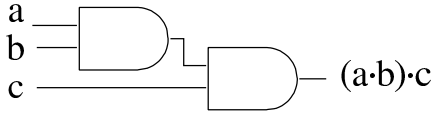
11. $a+(b+c) = (a+b)+c$

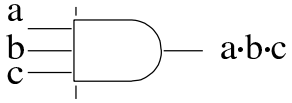




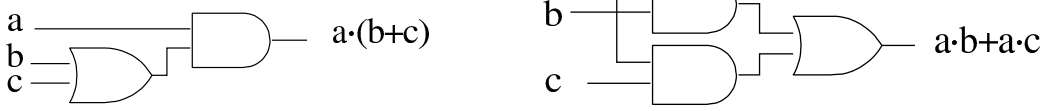
12. $a \cdot (b \cdot c) = (a \cdot b) \cdot c$



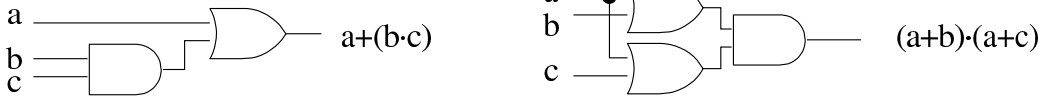




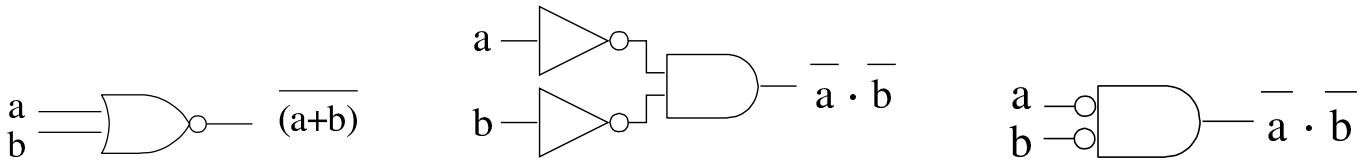
13. $a \cdot (b+c) = a \cdot b + a \cdot c$



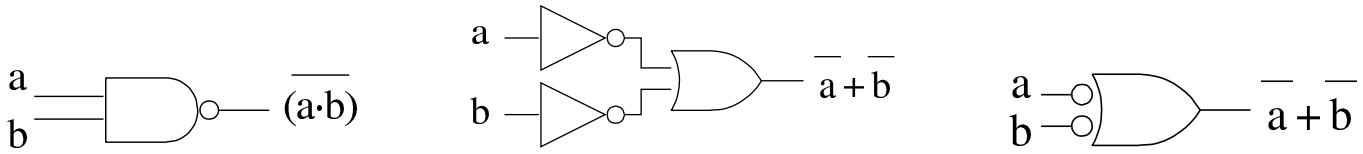
14. $a+b \cdot c = (a+b) \cdot (a+c)$



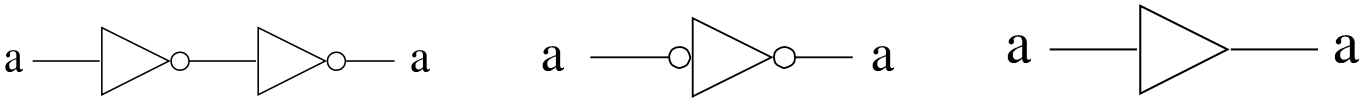
15. $\overline{(a+b)} = \overline{a} \cdot \overline{b}$



16. $\overline{(a \cdot b)} = \overline{a} + \overline{b}$



17. $\overline{(\overline{a})} = a$



When the logic function is defined using only the basic logic operators (NOT, AND, OR), the De Morgan's laws allow to transform easily a logic function into its complements. Given the function f it is possible to obtain the negated function $\overline{f}$ following a simple procedure:

- the original AND operators are replaced with OR operators and vice versa, using parentheses to keep the original evaluation order unchanged;
- all the variables are complemented (negated) with the NOT operator.

Example:

$$f = a + \overline{b} \cdot c + d \cdot \overline{e} + \overline{g} \cdot \overline{h}$$

$$\overline{f} = \overline{a} \cdot (b + \overline{c}) \cdot (\overline{d} + e) \cdot (g + h)$$

Please notice that $\overline{A}\cdot\overline{B}$ is not the same as $\overline{(A\cdot B)}$. So, writing $\overline{A\,B}$ might create confusion, because the line above seems to be unique.

1.10 Sum of produtcs

A logic function is an expression made of logic variables, logic constants and logic operators, which produces a logic result (true or false). However, a logic function can be described simply with the truth table that combines the input variable values with the expected result for every input combination.

Having a truth table it is possible to synthesize the corresponding logic function through the method known as the ***sum of products***. The sum of products is the 'sum' (logic OR) of all the ***fundamental products*** that describe every input condition. The following picture shows three truth tables with two, tree and four inputs; for every input combinations it is written the fundamental product, which is a function that becomes true only with a particular combination. For example, the fundamental product $\overline{A}\cdot B\cdot\overline{C}$ (or $\overline{A}\,B\,\overline{C}$) describes the combination A=0, B=1 and C=0.

Table 1.16. Fundamental products of functions having two, three and four input variables.

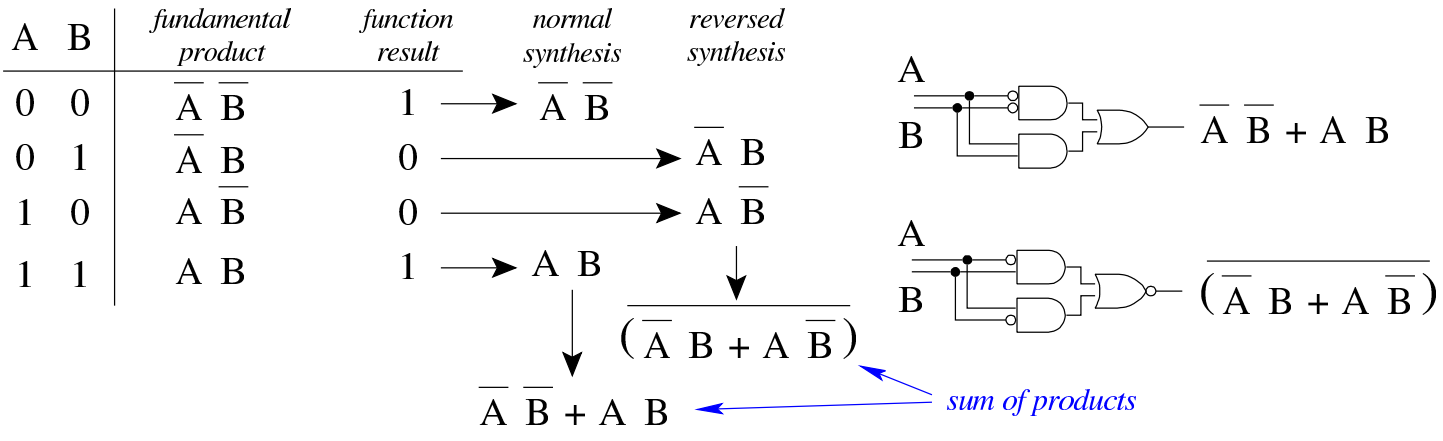
A	B	*fundamental product*
0	0	$\overline{A}\,\overline{B}$
0	1	$\overline{A}\,B$
1	0	$A\,\overline{B}$
1	1	$A\,B$

A	B	C	*fundamental product*
0	0	0	$\overline{A}\,\overline{B}\,\overline{C}$
0	0	1	$\overline{A}\,\overline{B}\,C$
0	1	0	$\overline{A}\,B\,\overline{C}$
0	1	1	$\overline{A}\,B\,C$
1	0	0	$A\,\overline{B}\,\overline{C}$
1	0	1	$A\,\overline{B}\,C$
1	1	0	$A\,B\,\overline{C}$
1	1	1	$A\,B\,C$

A	B	C	D	*fundamental product*
0	0	0	0	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$
0	0	0	1	$\overline{A}\,\overline{B}\,\overline{C}\,D$
0	0	1	0	$\overline{A}\,\overline{B}\,C\,\overline{D}$
0	0	1	1	$\overline{A}\,\overline{B}\,C\,D$
0	1	0	0	$\overline{A}\,B\,\overline{C}\,\overline{D}$
0	1	0	1	$\overline{A}\,B\,\overline{C}\,D$
0	1	1	0	$\overline{A}\,B\,C\,\overline{D}$
0	1	1	1	$\overline{A}\,B\,C\,D$
1	0	0	0	$A\,\overline{B}\,\overline{C}\,\overline{D}$
1	0	0	1	$A\,\overline{B}\,\overline{C}\,D$
1	0	1	0	$A\,\overline{B}\,C\,\overline{D}$
1	0	1	1	$A\,\overline{B}\,C\,D$
1	1	0	0	$A\,B\,\overline{C}\,\overline{D}$
1	1	0	1	$A\,B\,\overline{C}\,D$
1	1	1	0	$A\,B\,C\,\overline{D}$
1	1	1	1	$A\,B\,C\,D$

For example, the function known as NXOR (function of two input variables that is true only when the inputs have the same value) is synthesized with the procedure that is shown by the following figure.

Figure 1.17. NXOR function synthesis, starting from the truth table.

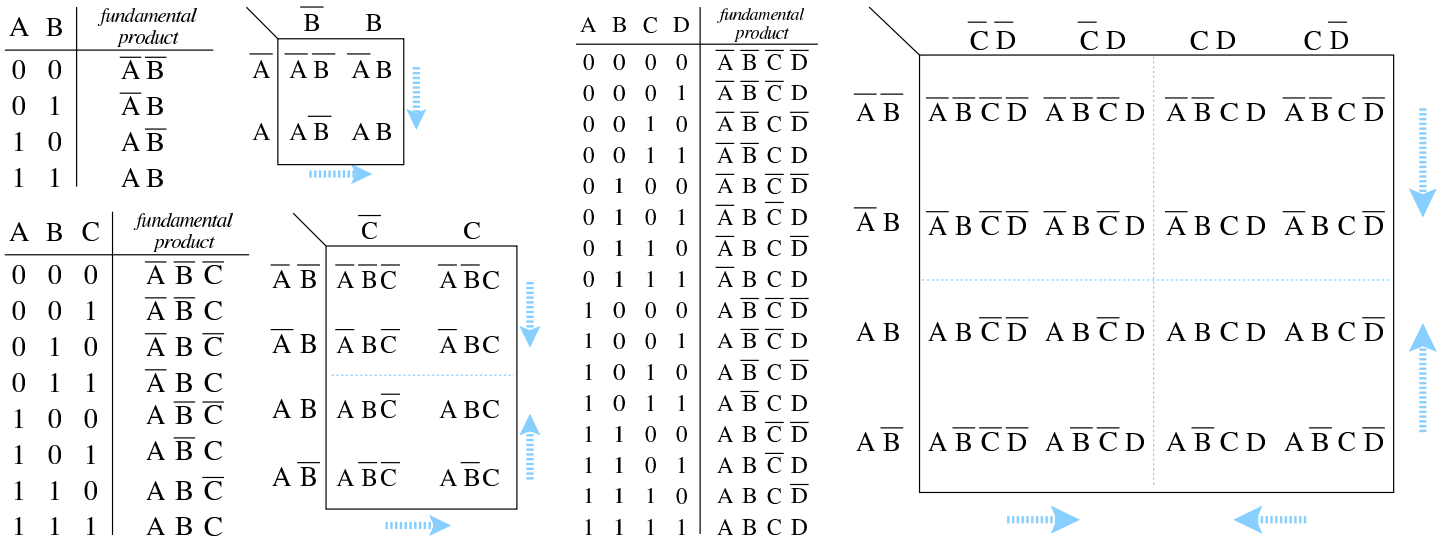


To synthesize a function, the fundamental products corresponding to the expected true combinations are added together (sum of products); as the figure shows, it is also possible to do the opposite to obtain the equivalent negated function.

1.11 Karnaugh maps

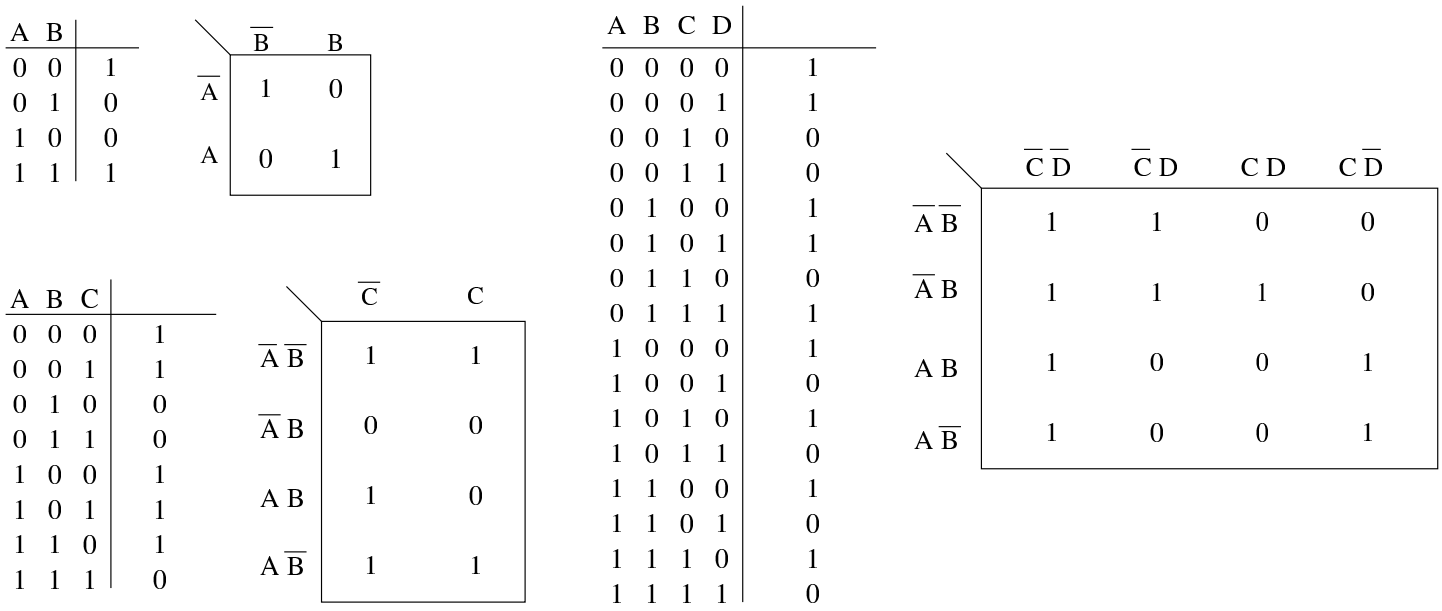
A logic function can be translated into a Karnaugh map, which can help to simplify the expression that synthesize the function. The Karnaugh map is a bidimensional representation of the fundamental products, from the truth table that is to be analyzed. In the following figures are shown three empty maps, for functions with two, three and four input variables: care must be taken to the fundamental produtcs sequence, because it does not correspond to the one used for the truth table.

Figure 1.18. Karnaugh maps containing the fundamental product definitions pertaining to each cell.



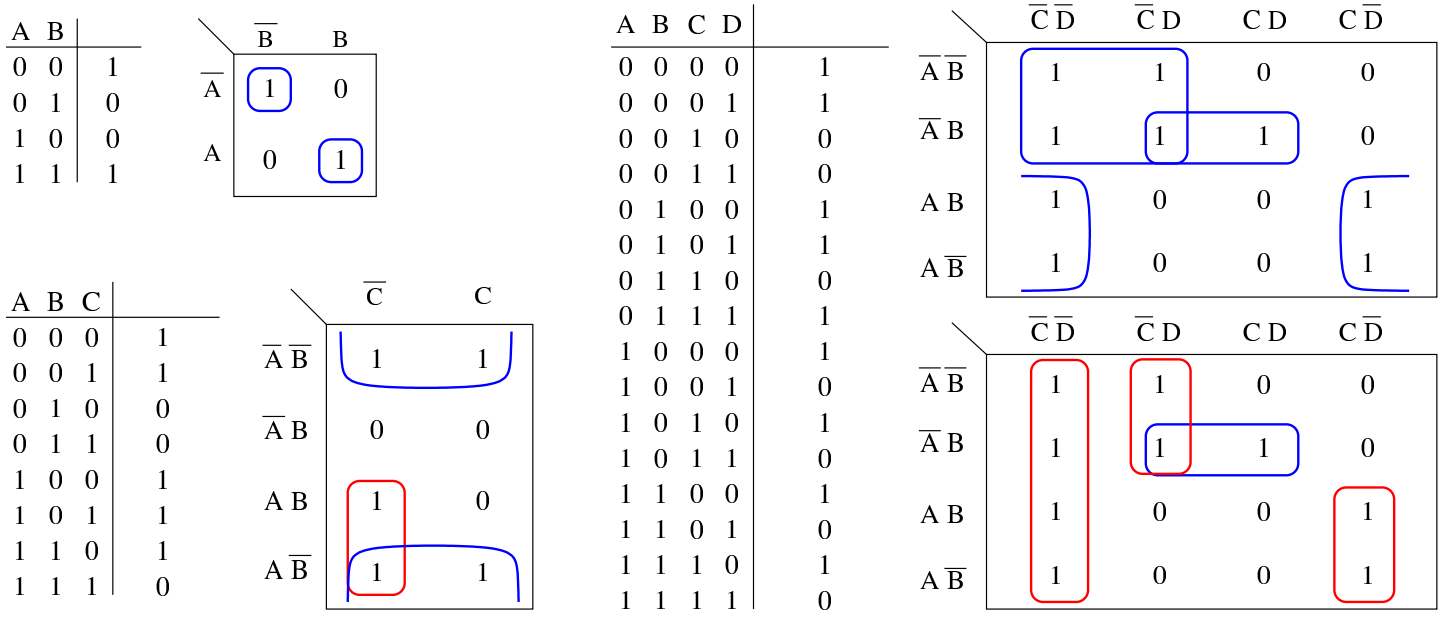
The maps should be used placing inside the cells a 1 where the fundamental products are valid. The following figures show some examples, matching each truth tables with the corresponding map.

Figure 1.19. Truth tables and Karnaugh maps pairs.



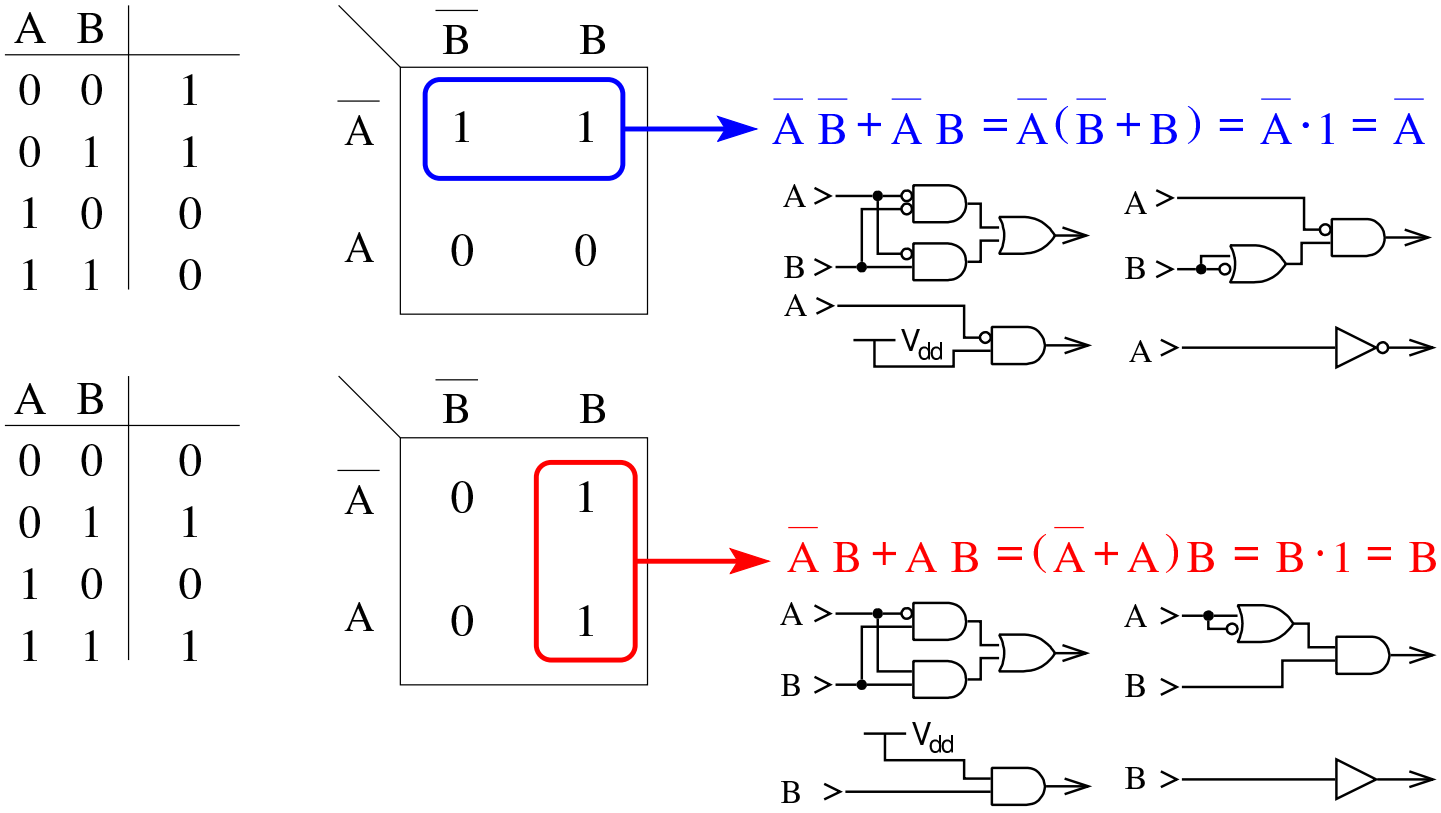
Once the map is prepared, the cells containing the value 1 should be grouped together in rectangular shapes, considering different grouping alternatives if possible. Please notice that these groups can be only of one, two, four, eight,... elements.

Figure 1.20. Karnaugh maps with the cells grouped horizontally and vertically; the map with four input variables shows two alternative groupings.



When two adjacent cells —vertically or horizontally— contain the value 1, one of the variable regarding the two cell is useless. The following figure demonstrates intuitively the process.

Figure 1.21. Simplification when two adjacent cells have the value 1.



The simplification comes out from the fact that x OR $\overline{x}$ is always true; therefore, when a rectangular group shows that the same variable appears both normal and negated, that variable can be simply ignored.

Figure 1.22. Example of a map that cannot be simplified.

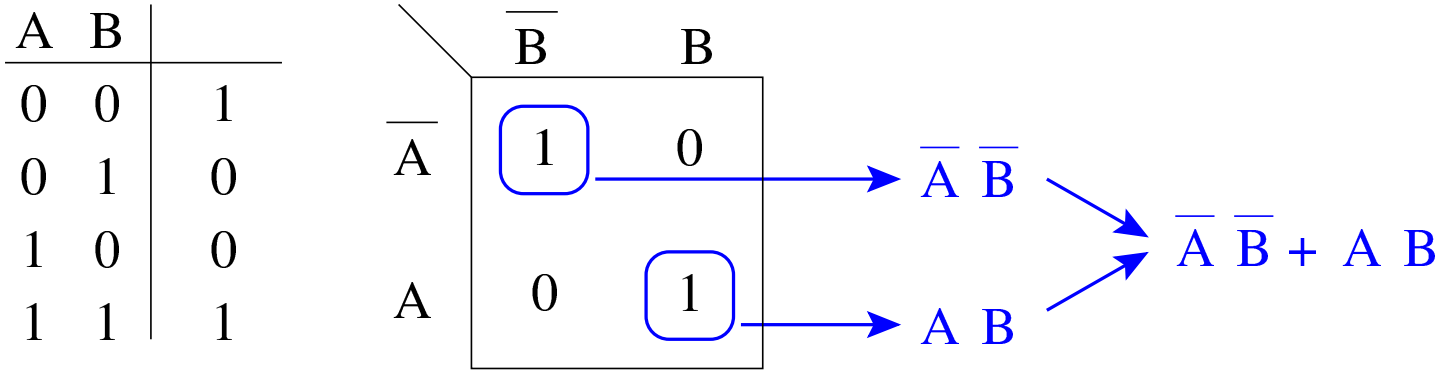


Figure 1.23. Example with three variables.

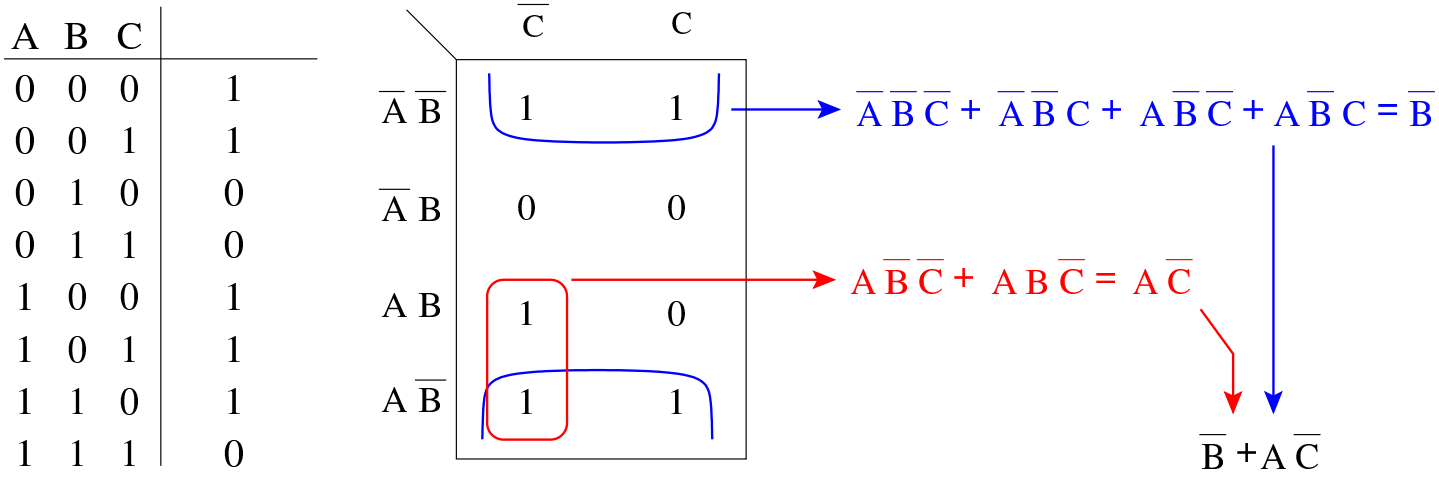
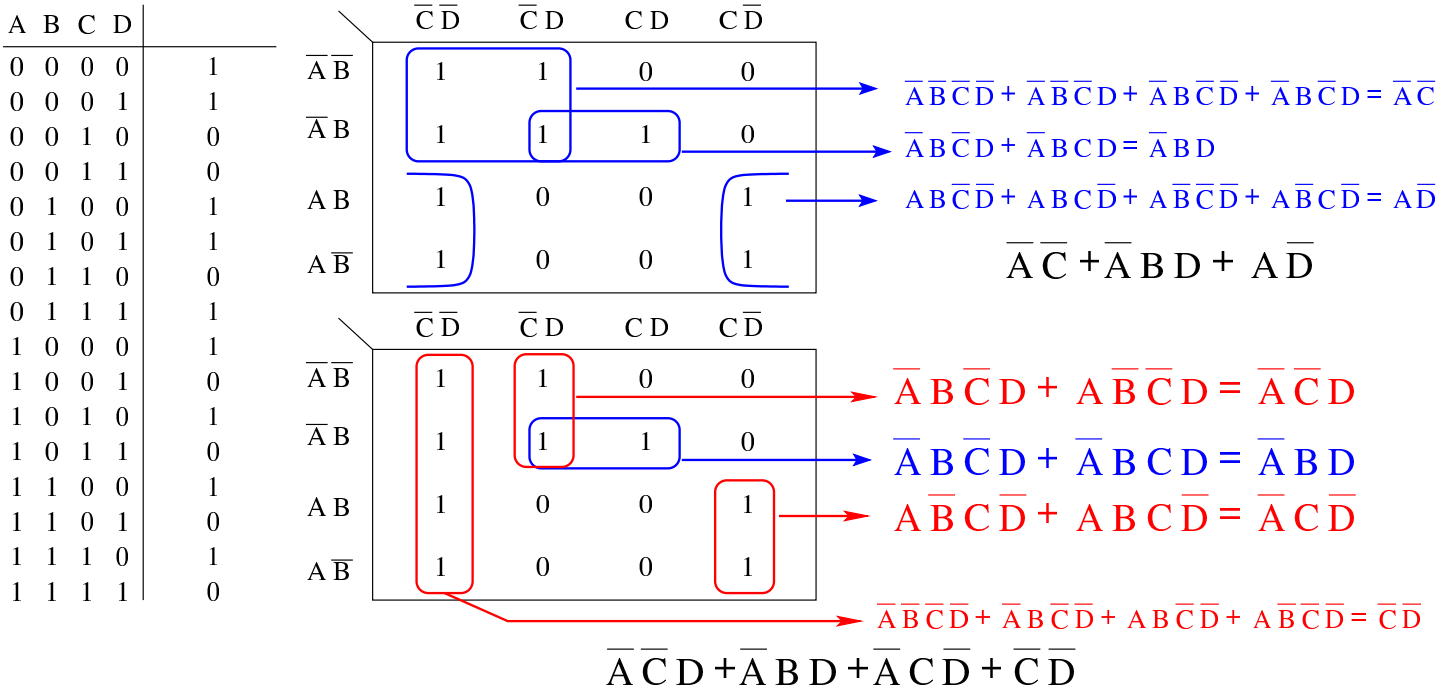


Figure 1.24. Example of alternative simplifications with a map having four variables.

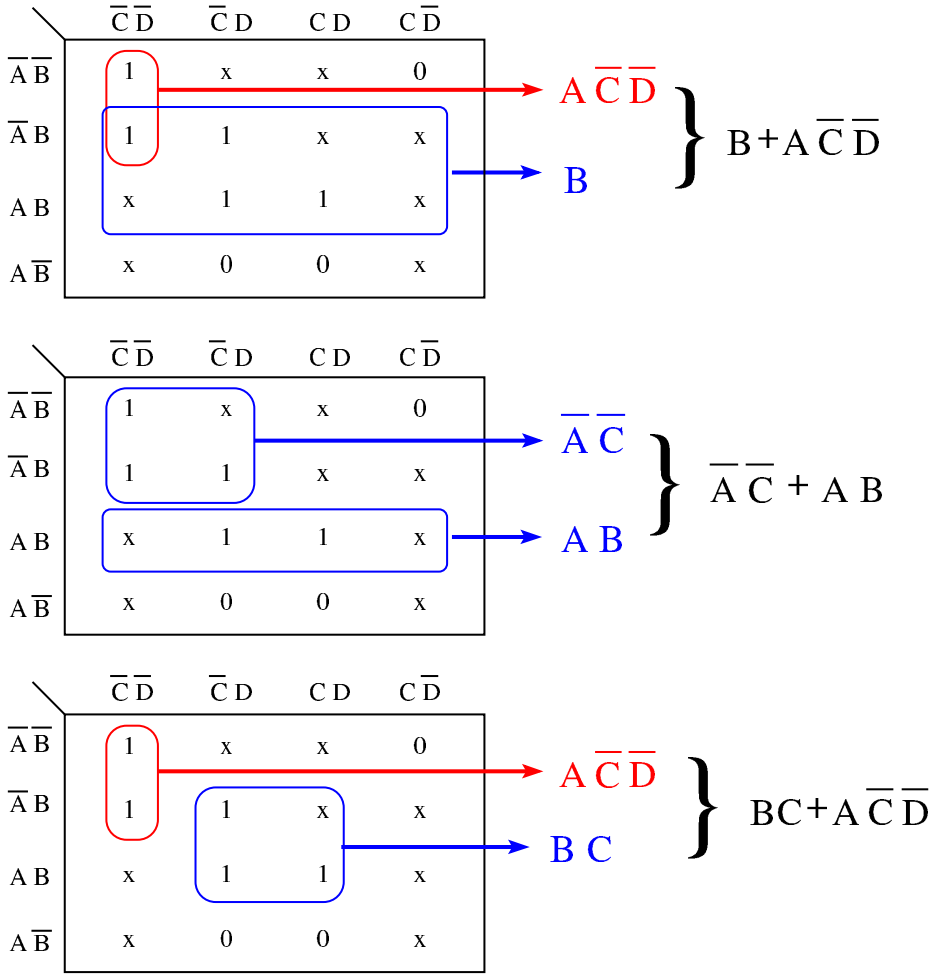


Karnaugh maps allow to simplify a logic function with no more than four variables, otherwise the map should have more than two dimensions and it would be impractical to view.

It might happen that the function output for some input combination is unspecified, because it doesn't matter what it is. When it happens, the synthesis can decide what value is better for the purpose of reducing the final expression. That is: it is free the decision whether these values should be grouped or not.

Figure 1.25. Example of alternative options when simplifying a function with some unspecified output values.

A	B	C	D	
0	0	0	0	1
0	0	0	1	x
0	0	1	0	0
0	0	1	1	x
0	1	0	0	1
0	1	0	1	1
0	1	1	0	x
0	1	1	1	x
1	0	0	0	x
1	0	0	1	0
1	0	1	0	x
1	0	1	1	0
1	1	0	0	x
1	1	0	1	1
1	1	1	0	x
1	1	1	1	1



Karnaugh maps can be used also for logic networks with more than one output for the same inputs; in that case, these networks are seen as multiple logic functions sharing the same input variables: it is probably useful to try to share also some more elements inside the two functions as it is shown in the following example.

Figure 1.26. Two functions of the same input variables, synthesized with two maps.

A	B	C	f_a	f_b
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	0

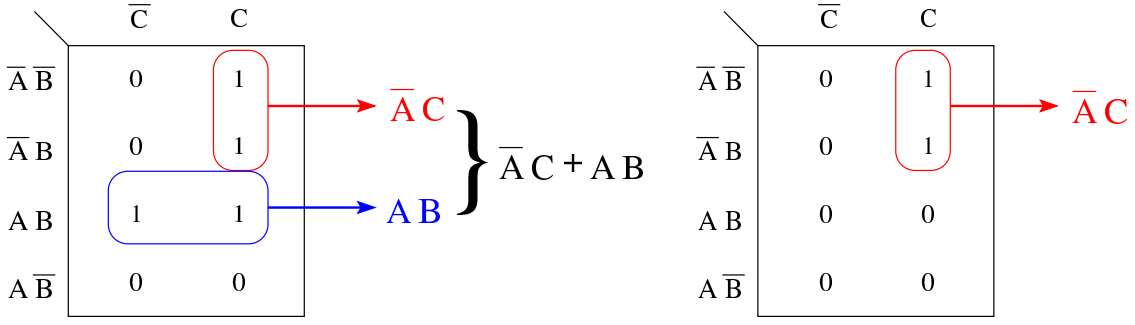
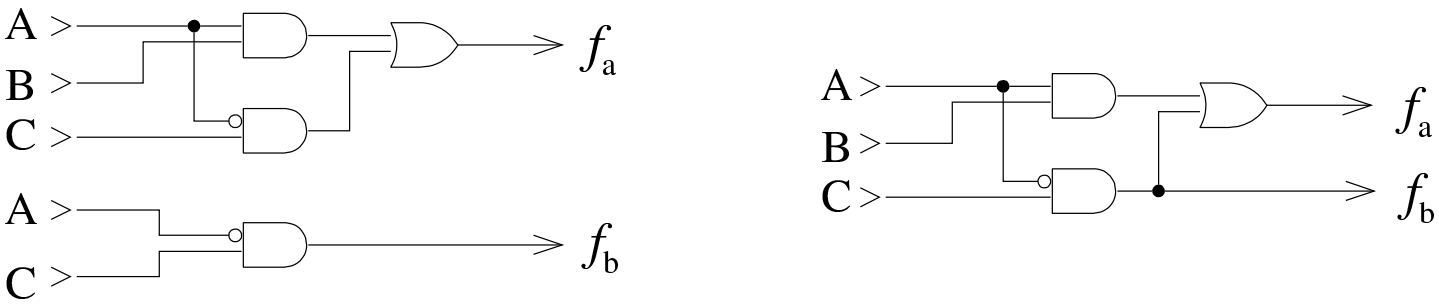


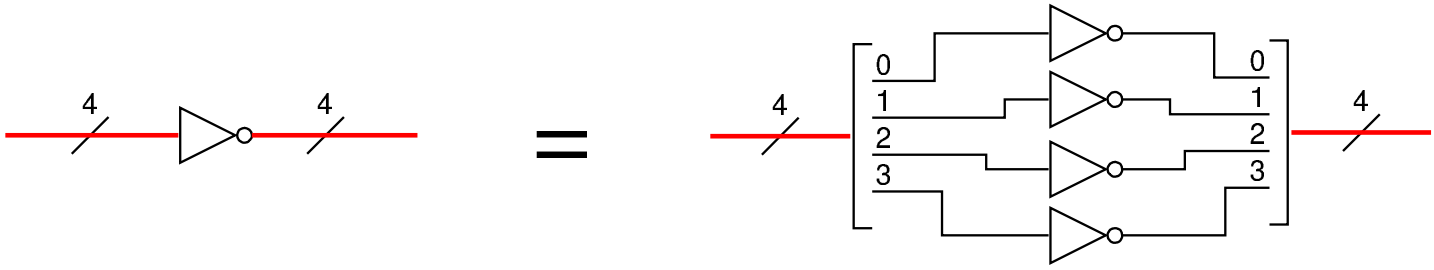
Figure 1.27. Logic network solution: divided on the left and joined on the right (saving a logic gate).



1.12 Logic gates and multiple lines

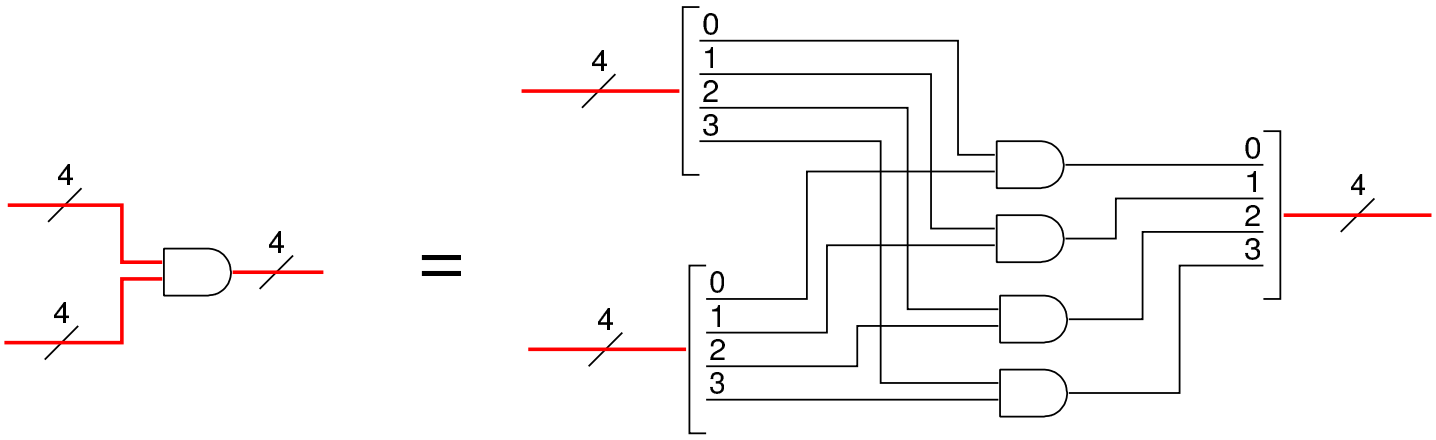
Designing a complex network it is possible to draw multiple lines together if the connected logic is replicated for each one.

Figure 1.28. Quad inverter.



The above example shows four inverters designed in paralel, using on the left a compact way, where the four input and output lines are joined together in a single multiple one. The following figure shows a similar paralel connection with four AND gates.

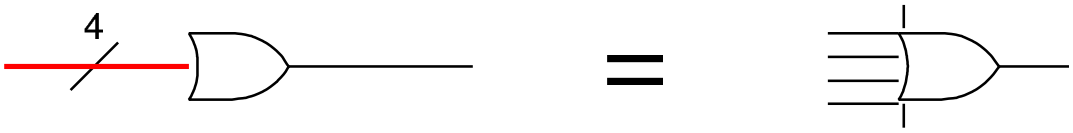
Figure 1.29. Quad AND.



The following figure shows a different kind of line joining, related to multiple input gates: the OR gate has four input lines, but on the left the four lines are joined in a

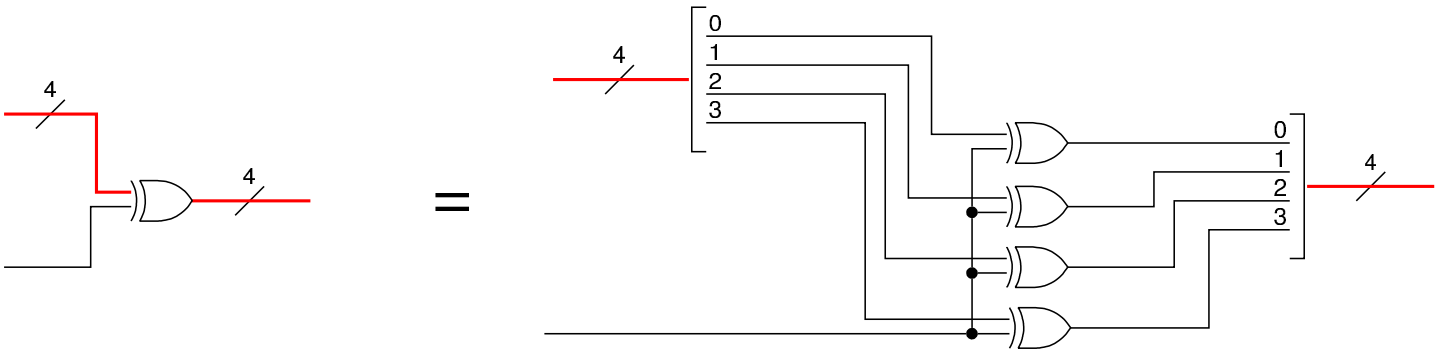
single multiple one.

Figure 1.30. Four input OR.



A paralel gate connection might require that some ports be connected together to a single line. The following figure shows four XOR gates with one of the two input ports connected to the same line.

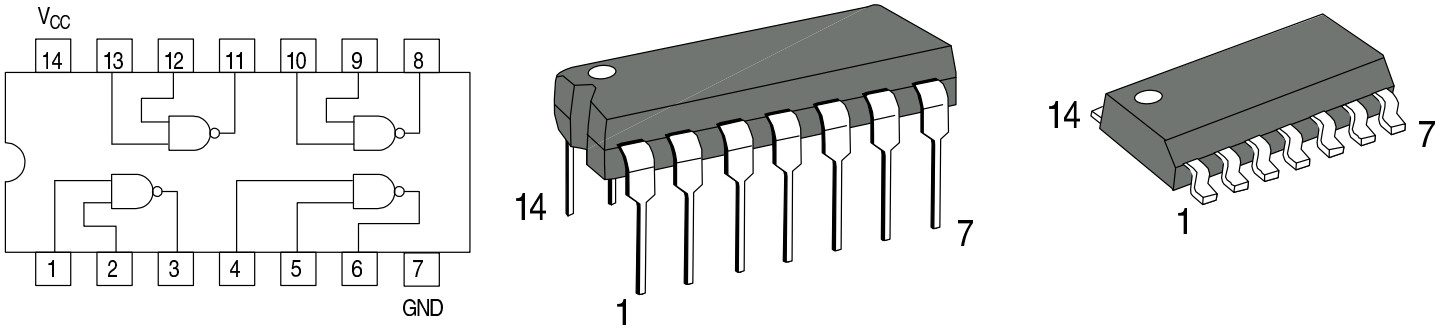
Figure 1.31. Quad XOR with a common input line.



1.13 Electronic logic gates

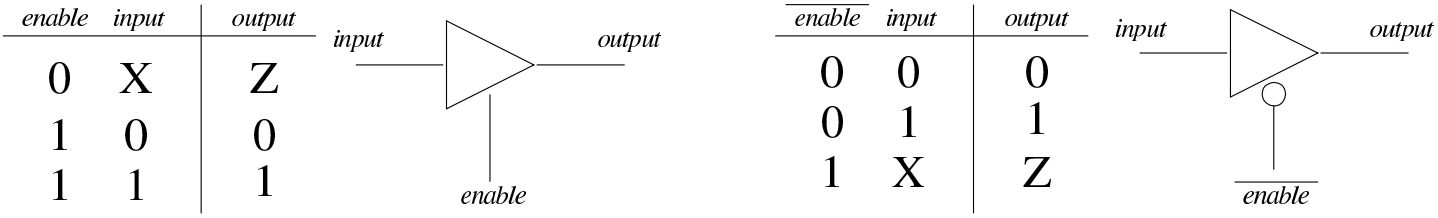
The logic gates are usually implemented inside integrated circuits (IC), with various technologies and densities. During the 1970's it was commonly used the so called ***discrete logic***, made of small integrated circuits with few logic gates or simple specific logic circuits. The old 'discrete logic' might give the idea of what can be concretely a logic gate in reality.

Figure 1.32. Four NAND gates inside an integrated circuit. The schematic on the left shows that the connections 7 and 14 must be used for the power supply: Vcc/Vdd and GND (ground).



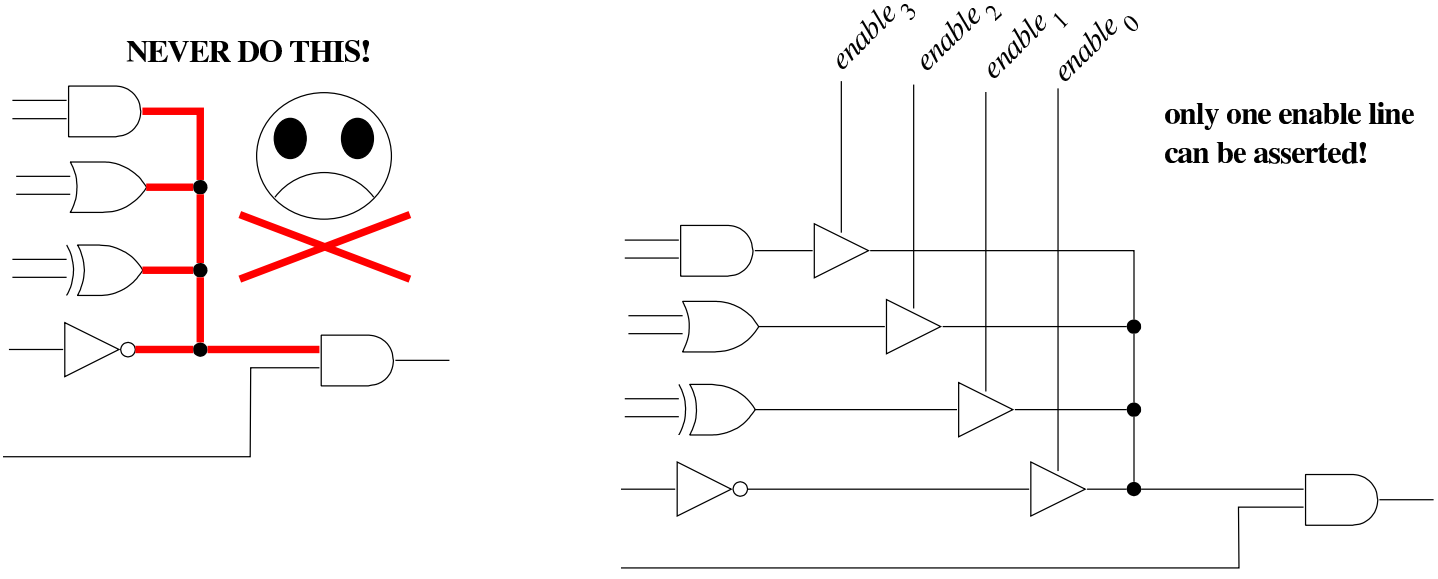
The boolean algebra consider only three logic states: 0, 1 and ‘X’, for ‘true’, ‘false’ and ‘unspecified’ (unknown). The electronic logic adds another state: ‘Z’ for ‘high impedance’ or ‘floating’.

Figure 1.33. Tri-state buffer. On the left side the ***enable*** line is ‘active high’; on the right side the $\overline{\textbf{\textit{enable}}}$ line is ‘active low’.



The tri-state buffer, shown above, works like a buffer, replicating the same value received from the input to the output, but only when the ***enable*** input is asserted, otherwise its output becomes virtually ‘disconnected’. The state of isolation that the tri-state buffer can have on the output, when the ***enable*** input is negated, is called ‘high impedance’ or ‘floating’ and is used the letter ‘Z’ to show it.

Figure 1.34. The connection of many output lines together is possible with tri-state buffers.

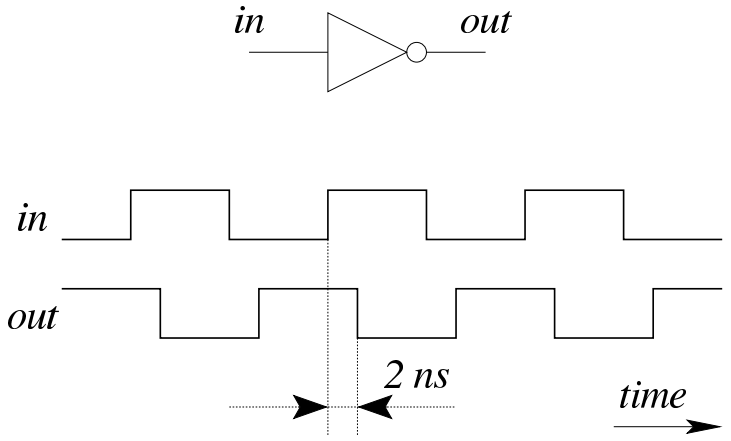


The figure above shows that it is not possible to connect together the output of two or more common logic gates, because it is not specified what should happen if these output drive a different logic state and because it will damage these components. On the right the circuit is modified with the addition of tri-state buffers; this time the connection is possible, but only under a strict condition: only one tri-state buffer can be enabled. But if no one of the four tri-state buffers is enabled, the last AND gate

on the right receives an unpredictable or confused input, corresponding to the Z state, which is neither 0 nor 1.

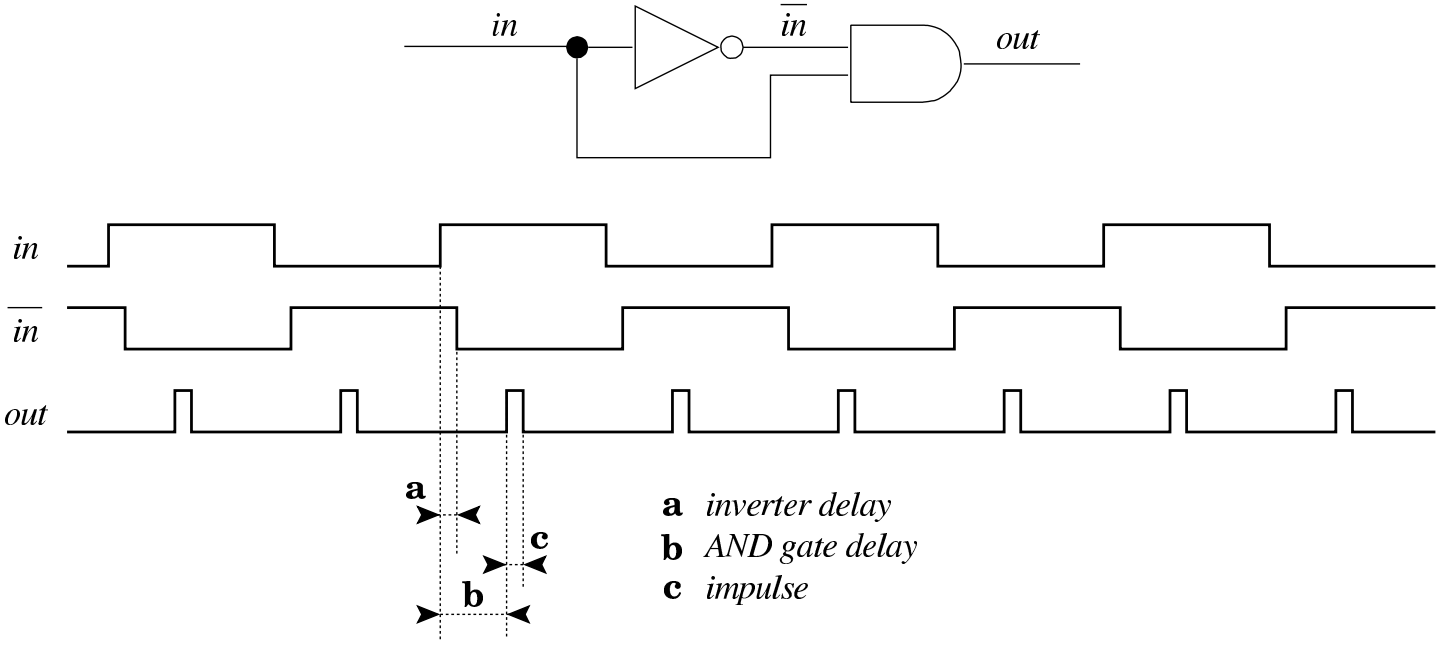
Electronic gates have also another important characteristic: the ***propagation delay***, which means that the gates output is always updated with a little delay, compared to the input changes.

Figure 1.35. Delay introduced by a simple inverter gate.



The figure above shows that even the simplest component, like the inverter gate, introduces a little delay. In that case it appear a delay of 2 ns. The following figure shows that the delay can be used to produce a short pulse.

Figure 1.36. Pulse signal produced with an inverter delay.



It should be noticed that the above example is not practical, because the delay is too short to be useful; a longer delay with a sequence of more inverters might solve the

problem.

1.14 HDL

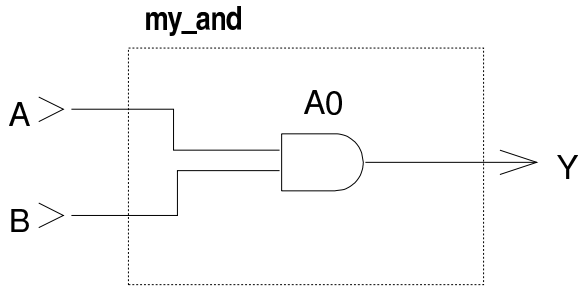
«

Logic networks can be simply designed in a way that can be read by human. When a logic network is to be treated as data, it should be 'designed' adding all the information needed and not just only the graphical aspect, whereas the graphical aspect might also be unnecessary and omitted. There are specific languages (just like the common programming languages) used to describe the logic networks and, more generally, the hardware. These are known as ***hardware description languages*** and the abbreviation HDL is commonly used.

There are two main HDL languages: VHDL and Verilog. Here is used Verilog because of its closeness to the C language, allowing to show examples without a previous detailed study of the language.

As a first example of Verilog coding it is shown how a simple AND gate could be synthesized. Verilog defines modules that are intended like black boxes with input and output ports, modules declared like they were functions; here the module is declared with the name ***my_and***, because the name 'and' is obviously reserved.

Figure 1.37. The module ***my_and*** that is to be synthesized with Verilog.



The first solution in Verilog code is very simple and it uses a 'continuous assignment' with a little delay (6 ns) to reproduce the propagation delay that real gates have.

Listing 1.38. Verilog code implementing the ***my_and*** module with a continuous assignment.

```
`timescale 1ns
module my_and (Y, A, B);
  input A, B;
  output Y;
  assign #6 Y = A & B;          // continuous assignment
```

```
endmodule
```

A very simple variation of the above example consists on the use of a gate instantiation, which is named ***A0***; like the previous example it is specified a 6 ns delay.

Listing 1.39. Verilog code implementing the ***my_and*** module with a gate instantiation.

```
`timescale 1ns
module my_and (Y, A, B);
  input A, B;
  output Y;
  and #6 A0 (Y, A, B);          // gate instantiation
endmodule
```

The two example above declare the module ***my_and*** and specifies that ***A*** and ***B*** are input ports (wires), whereas ***Y*** is an output port. Then, on the first example it is assigned to the ***Y*** wire the result of $A \cdot B$, with a 6 ns delay, whereas on the second example it is used a gate instantiation with the same delay. Please notice that the continuous assignment of the first example and the gate instance of the second just work continuously: every time that ***A*** or ***B*** change value, after the specified delay they update the value assigned to ***Y***.

A more complex solution might requires to use a memory, defining that the output line is the output of a virtual register with the same name. This virtual register works like a local variable of a single bit.

Listing 1.40. Verilog code implementing the ***my_and*** module with a propagation delay of 6 ns.

```
`timescale 1ns
module my_and (Y, A, B);
  input A, B;
  output Y;
  reg Y;                    // virtual register
  always @(A or B)          // do if A or B change value
    begin
      # 6;                  // propagation delay
```

```
        Y = A & B;          // assign the result to Y
    end
endmodule
```

The above code, after the module ports declarations, specifies that the ***Y*** wire comes from a register, so that the value of ***Y*** should be changed assigning something to it (inside a block). Then it start a block delimited by the keywords **`begin`** and **`end`**, which is executed every time that at least one of the input ports (***A*** and ***B***) changes its value. This block starts with a 6 ns pause, then the bitwise $\boldsymbol{A \cdot B}$ is calculated and assigned to ***Y***.

Chapter 2
Introduction to Verilog

2.1 Minimal notions 33
2.2 Hello world! 33
2.3 Compiler directives 38
2.4 Signal values 41
2.5 Literal constants 41
2.6 Data types 43
2.7 Ports 46
2.8 Parameters 48
2.9 Module declarations 48
2.10 Module instance 51
2.11 Fully qualified path names 51
2.12 Primitives 52
2.13 Expressions 54
2.14 Vector and array addressing 58
2.15 Strings 59
2.16 Functions 60
2.17 Netlist modules: combinational circuits 62
2.18 Behavioral modules 68
2.19 Procedural assignments 69
2.20 Blocking delays 71
2.21 Wait for a level event 72
2.22 Wait for an event expressions 74
2.23 Event variables 76
2.24 System tasks and functions 76

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

2.25 Control structures 78
2.26 Thread control 81

51 ! 54 != 54 !== 54 * 54 */ 33 + 54 . 33 / 54 /* 33 // 33 ; 33 == 54 === 54 ?: 54 addressing 58 always 68 and 52 array 43 58 assign 62 assignment 69 begin 68 behavioral module 68 bit range 58 bit size 43 58 blocking assignment 69 buf 52 bufif0 52 bufif1 52 case 78 casex 78 casez 78 combinational circuit 62 control structure 78 data type 43 defparam 51 delay 71 directive 38 end 68 endmodule 48 event 43 76 event expression 74 expression 54 floating point number 41 for 78 forever 78 fork 81 function 60 gate instance 52 HDL 31 if 78 initial 68 inout 46 input 46 integer 43 integer number 41 join 81 level event 72 literal constant 41 macro 38 memory 43 module 48 module instance 51 nand 52 net 43 netlist module 62 non-blocking assignment 69 nor 52 not 52 notif0 52 notif1 52 nxor 52 or 52 output 46 parameter 48 port 46 primitive 52 primitive instance 52 procedural assignment 69 real 43 reg 43 repeat 78 scalar 43 signal 41 string 41 59 supply0 43 supply1 43 switch instance 52 symbolic macro 38 system function 76 system task 76 thread 68 thread 81 time 43 top level module 48 tri 43 tri0 43 tri1 43 triand 43 trior 43 variable 43 vector 43 58 Verilog 31 wait () 72 wand 43 while 78 wire 43 wor 43 x 41 xor 52 z 41 & 54 && 54 @ 74 ^ 54 ~ 54 ~& 54 ~^ 54 ~| 54 | 54 || 54 {{}} 54 {,} 54 $display() 33 76 $finish 76 $finish() 33 $monitor() 76 $time 76 - 54 < 54 <= 54 << 54 <<< 54 > 54 >= 54 >> 54 >>> 54 % 54 `define 38 `else 38 `endif 38 `ifdef 38 `ifndef 38 `timescale 38

Logic networks can be simply designed in a way that can be read by human. When a logic network is to be treated as data, it should be 'designed' adding all the information needed and not just only the graphical aspect, whereas the graphical aspect might also be unnecessary and omitted. There are specific languages (just like the common programming languages) used to describe the logic networks and, more generally, the hardware. These are known as ***hardware description languages*** and the abbreviation HDL is commonly used. There are two main HDL languages: VHDL and Verilog. Verilog has the advantage to be more terse and similar to the C language.

When an HDL language is used, the project usually has two purpose: simulation and synthesis. Simulation is used to verify the project with a software tool; synthesis

is made to obtain some specific code for programmable hardware or to produce a final integrated circuit. When using an HDL language, some code might have specific meaning for the simulation that the synthesis phase might ignore.

To verify the examples of this chapter, the TKGate and Icarus simulation environments might be used (*http://www.tkgate.org* , *http://iverilog.icarus.com*).

2.1 Minimal notions

There is some similarity between Verilog and the C language, but Verilog is not C and the resemblance is limited to some aesthetics. The first minimal notions to start reading the Verilog code are the following:

- comments are the same as C (`/*...*/` and `//`);
- white spaces are ignored unless they are delimited as string constants;
- the semicolon (`;`) is used to terminate the statements and the comma (`,`) is used to separate elements inside a list;
- identifiers (the names used to identify components defined inside the code) are case sensitive and should start with a letter or the underscore (_) and might contain letters, digits, underscore (_) and dollar (`$`);[1]
- all Verilog keywords are lowercase.

2.2 Hello world!

A first minimal program is useful to test the simulation environment. The Verilog language includes some functions and instructions whose name start with dollar (`$`), functions and instructions that are intended to control the simulation environment. The typical simulation environment should have the ability to display some text that Verilog can output with the **`$display()`** function, which works in a similar way to the **`printf()`** found in C.

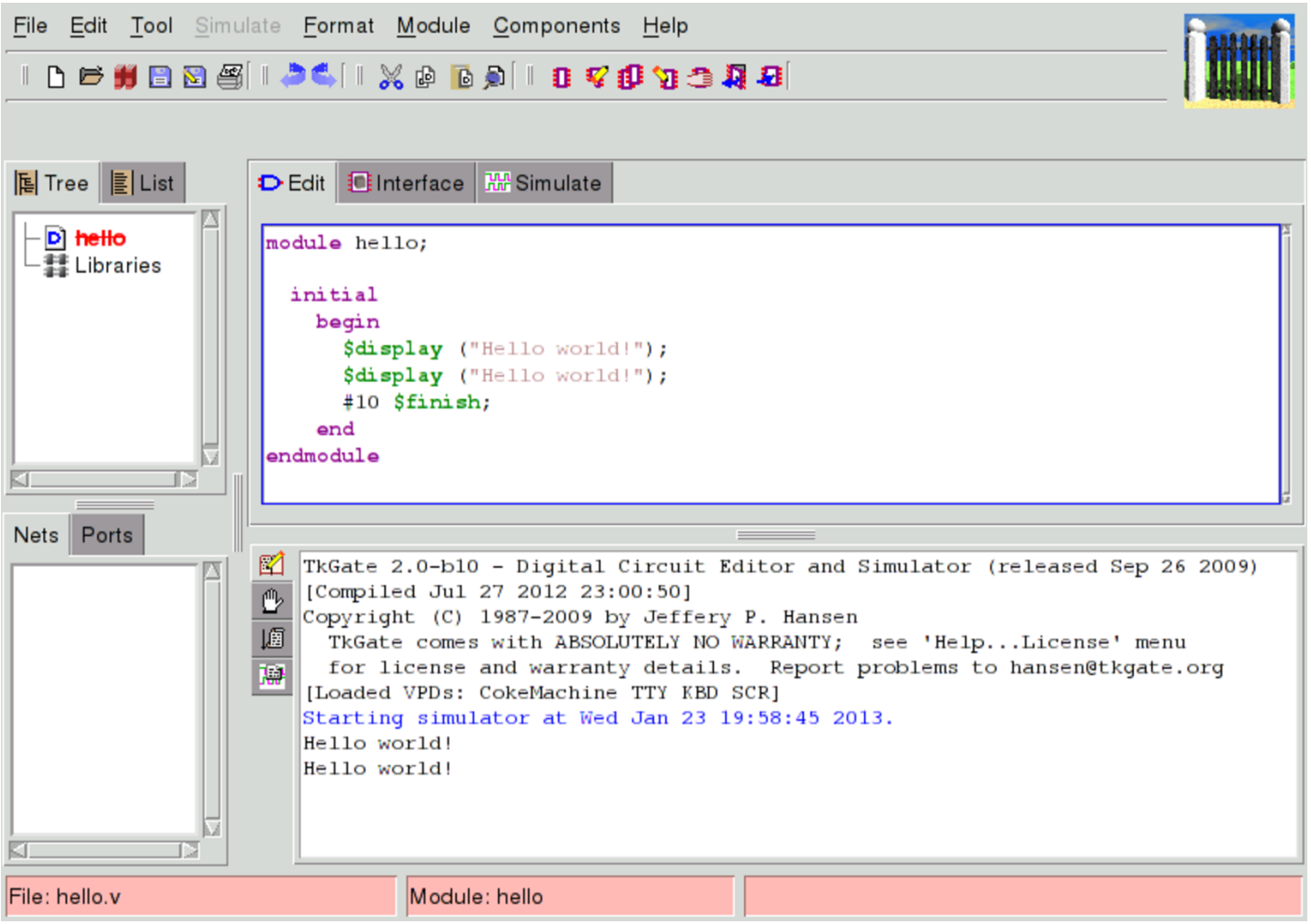
Listing 2.1. Two times “Hello world!”.

```
module hello;
  initial
    begin
      $display ("Hello world!");
      $display ("Hello world!");
      #10 $finish;
    end
endmodule
```

The example listed above shows two times the text “Hello world!”, then it waits 10 time units and then it stops the simulation with the **`$finish`** instruction. Please notice that the **`$display()`** function adds a new-line at the end of the string and the double display is intended to show this feature. It is supposed that the file containing the above listing is named ‘`hello.v`’; to run TKGate with it, the following command might be used:

`$` **`tkgate20 hello.v`** [*Enter*]

Figure 2.2. TKGate after the program end.



The above example might be also compiled and tested with Icarus Verilog in the following way:

`$` **`iverilog -o hello.vvp hello.v`** [*Enter*]

The above command generates the file '`hello.vvp`' that can be executed by the command **`vvp`**:

`$` **`vvp hello.vvp`** [*Enter*]

```
Hello World!
Hello World!
```

This video shows how to replicate the above example, with TKGate and with Icarus Verilog, starting from the same source '`hello.v`': ogv *http://www.youtube.com/watch?v=kGofhPGWErc* .

To use a Verilog simulation environment it is also necessary to be able to trace a wave at some point, to see dynamically what happens inside some virtual wires. The next example is made to generate two different clock signals at the variables ***x*** and ***y*** (these variables represent a wire with memory and it doesn't matter how it should be implemented).

Listing 2.4. Two different clock signals.

```
module wave;
  reg x;
  reg y;
  initial
    begin
      $dumpfile ("wave.vcd");
      $dumpvars (0, wave);
      x = 0;
      y = 0;
    end
  always
    begin
      # 20;
      x = 1;
      # 20;
      x = 0;
    end
  always
    begin
      # 15;
      y = 1;
      # 15;
      y = 0;
    end
endmodule
```

The above listing contains two simulation functions necessary to Icarus Verilog to create the '`wave.vcd`' with all the variables variations; this file is then read by GTK-

wave. The following command assumes that the above listing is contained inside the file ‘`wave.v`’:

```
$ iverilog -o wave.vvp wave.v [Enter]
```

```
$ vvp wave.vvp [Enter]
```

The simulation does not stop by itself and it is necessary to send an interruption signal with the keyboard combination [*Ctrl c*].

```
$  [Ctrl c]
```

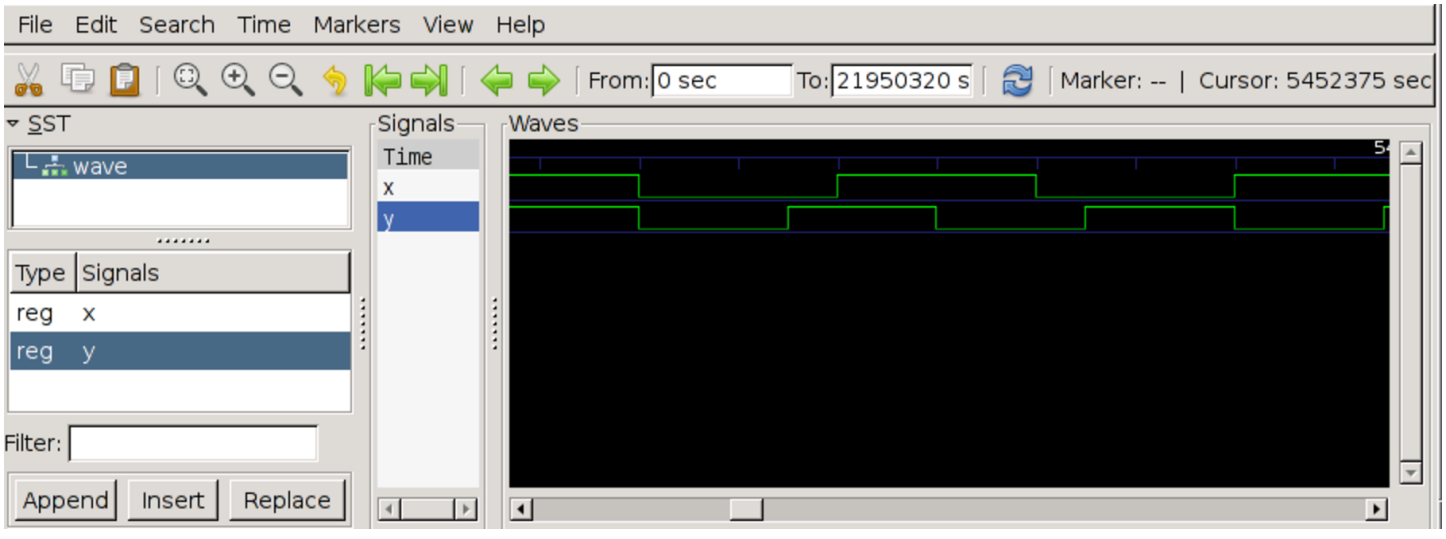
Then **`vvp`** turns to work in interactive mode and it is necessary to type the command **`finish`**:

```
> finish [Enter]
```

At this point there is the file ‘`wave.vcd`’ and it can be read with GTKwave:

```
$ gtkwave wave.vcd [Enter]
```

Figure 2.5. GTKWave with the variable ***x*** and ***y*** selected.



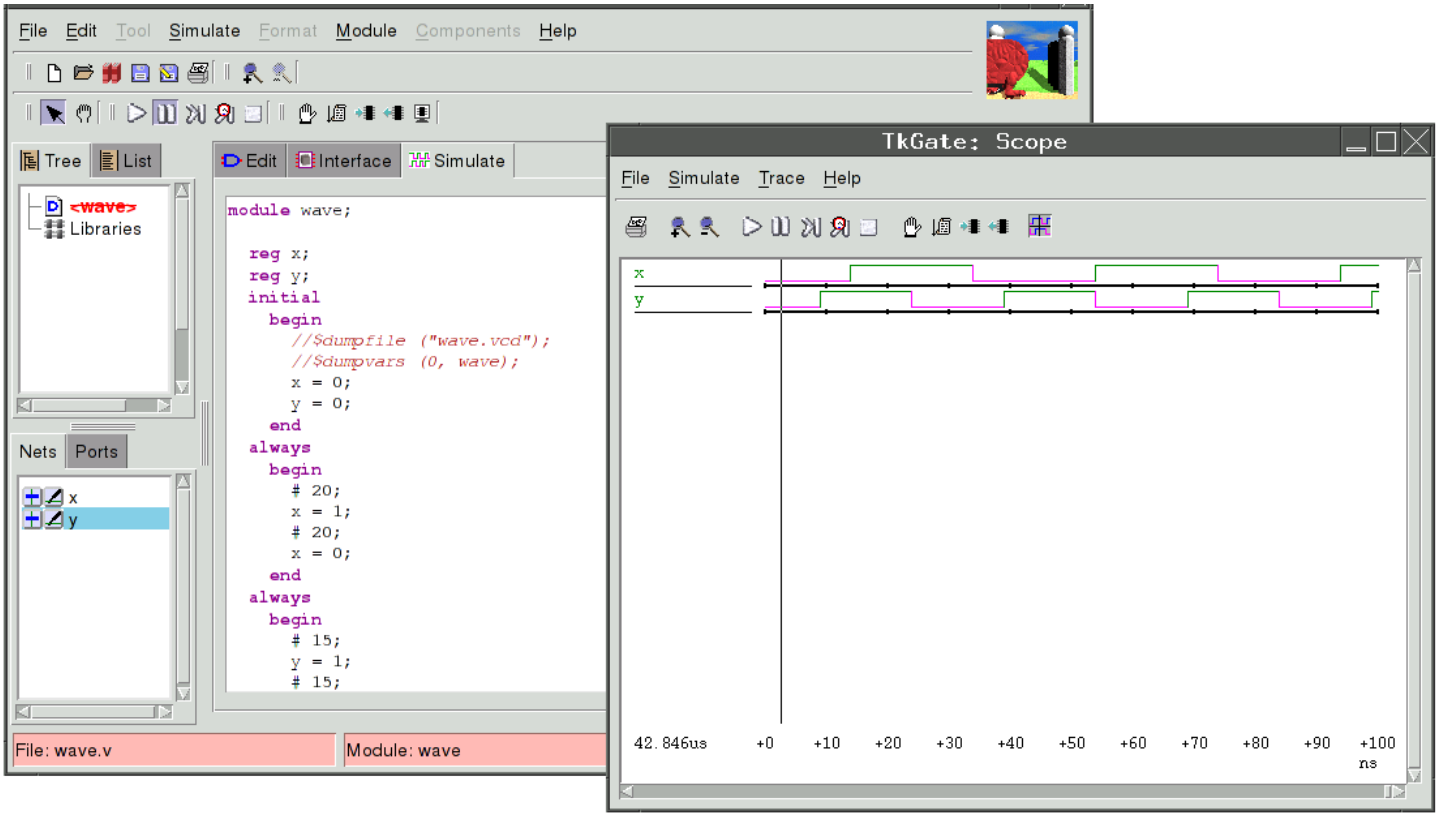
The picture above shows GTKWave with the variables ***x*** and ***y*** already selected. To do so, on the left the module name (**`wave`**) must be selected, then the variable names appear just below: the variable names must be dragged and dropped into the black window that should show the waves.

TKGate can use the same source example, but it does not know the functions ***$dumpfile()*** and ***$dumpvars()***, so these lines must be removed or commented.

```
$ tkgate20 wave.v [Enter]
```

The following picture shows TKGate running the simulation with a 'scope' window showing the waves, selected from the net list on the left.

Figure 2.6. GTKWave with the variable ***x*** and ***y*** selected.



This video shows how to replicate the above example, with Icarus Verilog and TKGate, starting from the same source '`wave.v`': ogv *http://www.youtube.com/watch?v=NLzcXuadiIQ* .

2.3 Compiler directives

«

The C language has a pre-compiler that looks for directives like **`#define`** and **`#ifdef`**. The Verilog language has similar directives that starts with the character **`** (back quote). Like in the C language, the compiler directives do not have the final semicolon.

Like in the C language there is a **`define** directive to declare a symbolic macro:

```
`define macro replacing_text
```

For example, the symbolic macro **`MY_MACRO** representing the value 45 could be declared as:

```
`define MY_MACRO 45
```

To use the symbolic macro inside an expression, it must be used with the ` prefix, like this:

```
x = a + `MY_MACRO;
```

Like in the C language there are the directives **`ifdef**, **`ifndef**, **`else** and **`endif**, to conditionally select the Verilog code.

```
`ifdef macro
  code_if_macro_exists
`else
  alternative_code
`endif
```

```
`ifndef macro
  code_if_macro_does_not_exist
`else
  alternative_code
`endif
```

See the following example:

```
`ifdef MY_MACRO
  x = a + `MY_MACRO;
`else
  x = a;
`endif
```

In the above example, the variable ***x*** is the sum of ***a*** and the macro **`MY_MACRO**, but only if the macro itself is already declared. The next example is just the opposite, but with the same meaning:

```
`ifndef MY_MACRO
  x = a;
`else
  x = a + `MY_MACRO;
`endif
```

The Verilog language requires the definition of the ***timescale***, which is the minimal time unit used for the simulation. The timescale is defined with the **`timescale** directive:

`timescale ***integer_delay_unit*** [/ ***max_resolution***]

The integer delay unit time and the max resolution are specified with values that should be 1, 10 or 100 followed by “s” (seconds), “ms” (milliseconds), “us” (microseconds), “ns” (nanoseconds), “ps” (picoseconds) or “fs” (femtoseconds).

```
`timescale 10ns/1ns
```

The above example requires that the simulation is made with a precision of 1 ns, whereas the integer delays are 10 ns multiples, but this means that a delay of 1 is 10 ns long and a decimal delay time can be specified, up to the first decimal position. For example, a delay of 1.3 is equal to 13 ns, but 1.35 is also equal to 13 ns, because the minimal resolution is of 1 ns. The following example specifies that only integer delays notations are allowed (1 ns):

```
`timescale 1ns
```

2.4 Signal values

Verilog consider four ‘logical’ states: 0, 1, unknown and floating (hi-impedance). The states 0 and 1 are the well defined logic values; if the logic value is unknown or it is not possible to establish it, the symbol **`x`** or **`X`** can be used to express it. If a wire becomes isolated from the other connections, the symbol **`z`**, **`Z`** or **`?`** can be used to describe it.

`0`	logical 0	`x` | `X`	unknown or unspecified
`1`	logical 1	`z` | `Z` | `?`	floating (high impedance)

2.5 Literal constants

A ***literal constant*** is a constant value represented directly by its value. There are three types of literal constants that Verilog can expect: integer numbers, floating point numbers and strings.

The literal constant representing an integer number inside Verilog is much different from the common programming languages, because it can contain the bit-range, or size:[2]

[[*size*] **'** *base*] *value*

[[*size*] **'** `b` | `o` | `d` | `h`] *value*

The ***base*** is a letter specifying the base in which the following number is represented, but it is always preceded by an apostrophe. The allowed bases are: **`'b`**, binary; **`'o`**, octal; **`'d`**, decimal; **`'h`**, hexadecimal. Here are some examples with size and base specified:

`7'd32` – 7-bit 32_{10}

7'd232 – 7-bit 104_{10}[3]

15'h3B4F – 15-bit $3B4F_{16}$

5'b11101 – 5-bit 11101_2

If the size of the value is not specified, a default size is used, but it depends on the implementation and it should be at least of 32 bits (although it is not guaranteed). Here are some more examples without size specification:

'd32 – 32_{10}

'd232 – 232_{10}

'h3B4F – $3B4F_{16}$

'b11101 – 11101_2

If even the base is not specified, it is intended to be a decimal value.

A literal constant can contain some underscore characters, which are ignored and are useful only to separate the digits for some aesthetical reasons:

16'b0101_0100_1110_1011 – 0101010011101011_2

The integer representation shown above is related to unsigned numbers, which might be extended as unsigned numbers. The current Verilog standard allows also to specify signed numbers, but the handling of signed numbers is not simple and it is useful to avoid it in the beginning.

When the base is not decimal, the number can contain also unknown or floating values, but these particular values are extended if not all the requested bits are specified:

16'h1x2x – $0001xxxx0010xxxx_2$

16'hx2x – $xxxxxxxx0010xxxx_2$

8'b10zz10 – $0010zz10_2$

8'bzz10 – $zzzzzz10_2$

A floating point literal constant is represented as a decimal value with decimal point; for example: **0.123**, **1.23**, **12.34**. It is the implementation that decides in which way it is encoded internally and the bit size that it should have.

Strings are represented in double quotes. Each character inside the string occupies 8 bits and is encoded following the ASCII standard. For example: a string made of three characters requires 24 bits. Strings can contain a subset of the C language escape sequences: **\n**, **\t**, ****, **\"** and ***\ooo*** (the last escape sequence is a character defined by the octal code).

2.6 Data types

The Verilog language has many different 'things' that can represent data, divided into three main categories: ***nets***, ***variables*** and ***events***. Nets are just wire connections that can be driven with some value, but they cannot store it; variables are components (connected with wires) that can hold and keep a value; events are special variables used to hold the triggering of an event.

Net types represent wires that can have a value only when they are driven by some output. If a wire is driven by more than one source with different values, then there is a collision and the result on the wire is different, depending on the particular net type.

Table 2.14. Net types.

Type	Description
`wire`	A simple wire used to connect components: collisions result in unknown value.
`wand`	A wired AND net: collisions result in the AND of the values driven to the net.
`wor`	A wired OR net: collisions result in the OR of the values driven to the net.
`tri`	Just the same as **`wire`**, emphasizing the fact that there can be also the floating state.
`tri0`	A wire where a floating state is changed into 0.
`tri1`	A wire where a floating state is changed into 1.
`triand`	Just the same as **`wand`**, emphasizing the fact that there can be also the floating state.
`trior`	Just the same as **`wor`**, emphasizing the fact that there can be also the floating state.

Type	Description
`trireg`	A wire connected to a virtual capacitor, which can retain the last value if it is driven by a floating state.
`supply0`	A wire directly connected to the ground, resulting always into the value 0.
`supply1`	A wire directly connected to Vcc (or Vdd), resulting always into the value 1.

Variable types represents registers and other type of containers to which a value can be assigned.

Table 2.15. Variable types.

Type	Description
`reg`	A variable that can be used to represent a hardware register.
`integer`	A 2's complement signed integer variable not intended to represent a hardware register, which should have at least a 32-bit size.
`real`	A floating point (signed) variable not intended to represent a hardware register, which should have at least a 32-bit size.
`time`	An unsigned integer variable, with at least a 64-bit size, useful for saving the time counted as a number of simulation time units. It is used to hold the value returned by the ***$time*** system variable.

The declarations of nets and **`reg`** type variables can be related to a single bit or to a numbered range of bits (the types **`integer`**, **`real`** and **`time`** have an implied bit size). The following examples show the declarations of nets and registers with a single bit size:

```
wire w1, w2, w3;   // w1, w2, w3 and w4 are nets of type
wire w4;           // 'wire'.
supply0 w5, w6;    // w5 and w5 are nets of type 'supply0'.
supply1 w7;        // w7 is a net of type 'supply1'.
reg r1, r2;        // r1, r2 and r3 are variables of type
reg r3;            // 'reg'.
```

To declare a 'word' of more than one bit, a range of bits is added before the name of

the net or register. This range is enclosed in square brackets, as the following syntax should explain:

[***msb*** : ***lsb***]

For example, the following declaration defines the 16-bit wires **`w16a`** and **`w16b`**:

```
wire [15:0] w16a, w16b;
```

For a better compatibility, the range should always start from bit zero, up to ***size***−1, like the example above.

When a net or **`reg`** is declared as a single bit, it is known as a 'scalar', whereas when it is declared with a bit size it is known as a 'vector'.

A net or variable, either scalar or vector, might be declared as an array, adding a range of elements in square brackets after the name, and if the array is of type **`reg`** it is also called a ***memory***:

name [***index_1*** : ***index_n***]

The following example shows a memory made of an array of register vectors:

```
reg [7:0] mem[0:255];
```

The memory ***mem*** is made of 256 8-bit cells, where the first cell is reached with the index 0 and the last with the index 255.

For a better compatibility, the index range should always start from cell zero, up to ***size***−1, like the example above.

To access an array element an index is used, in square brackets, just like the C language does. The index might be a literal constant or an expression that produces an integer value.

Table 2.19. Event type.

Type	Description
`event`	A variable that can be set (triggered) to represent an event.

The event type is a single particular one that is used to store the triggering of an event. This kind of variable is read only inside an event expression and when it is read it loses the trigger previously stored. At the moment it is useful to know just how an event variable is declared; the following example shows the creation of the event variable ***e***:

```
event e;
```

2.7 Ports

Almost all the Verilog code must be contained inside a module. A module is something like a box connected to the external world through ***port***s. So, a port is a connection that can be used to get data into the module (**`input`**), to put data out of the module (**`output`**) or to get and put data (**`inout`**).

When declaring a new module the port list should appear in parenthesis, but then it must be specified the direction of these ports. The example below shows the declaration of the module **`my_module`** with three ports, named ***a***, ***b*** and ***c***, used respectively for input, output and both:

```
module my_module (a, b, c);
  input a;
  output b;
  inout c;
  …
endmodule
```

As it can be seen from the example above, the keywords **`input`**, **`output`** and **`inout`** are used to declare a port, with its direction. But ports might have a size bigger than a single bit, in that case, the size is specified in the same way as for nets and registers.

```
module my_module (a, b, c);
  input [3:0] a;
  output [2:0] b;
  inout [1:0] c;
  …
endmodule
```

The above example shows that all the ports are vectors of various sizes. Please notice that there are no port arrays.

A port is a connection that, inside the module, might work as a net or as **reg**, but this fact should be specified, otherwise it is assumed that the internal connection is seen as a type **wire**.

```
module my_module (a, b, c);
  input [3:0] a;
  output [2:0] b;
  inout [1:0] c;
  reg [2:0] b;
  …
endmodule
```

The above example shows that the port ***b*** is driven by the module with a memory (might be a register). The same declaration might be done at the same time of the port declaration, in a more elegant way:

```
module my_module (a, b, c);
  input [3:0] a;
  output reg [2:0] b;
  inout [1:0] c;
  …
endmodule
```

2.8 Parameters

Verilog allows to declare symbolic constant, known as ***parameters***. The parameter declaration must specify also the value that it represents, with a literal constant or with an expression where literal constants and other parameters can be used.

```
module …;
  …
  parameter DLY = 3;
  parameter XDLY = 3 + DLY;
  …
endmodule
```

The above example shows the declaration of the parameter ***DLY*** with the value 3 and the parameter ***XDLY*** with the value 6.

The parameters allow to define 'parameterized' modules, so that, for example, the size of a net or **`reg`** can be defined when a module is instantiated. These details are described in the following sections, about module declaration and module instantiation.

2.9 Module declarations

Verilog requires that any circuit or procedure be contained inside a module. There must be at least a ***top level module*** that might use other modules. Verilog treats modules as classes of an object-oriented languages and, except for top level modules, the other modules are used after their instantiation. Please notice that any module that is never instantiated is a top level module.

```
module name [ (port_name [, ...]) ];
   [declarations]
   [assign assignment] ...
   [primitive_instantiation] ...
   [module_instantiation] ...
   [initial block] ...
   [always block] ...
endmodule
```

The syntax above shows the declaration of a module: it is important to notice the position of the semicolon and the absence of it after the keyword **endmodule**.

A module has usually ports, to communicate with the outside, except for the root module that does not requires them. If ports are present, their name should be listed inside parentheses, but the listed ports must also be declared specifying their direction and their data type.

```
module my_module (a, b, c);
  input [7:0] a;
  output reg [7:0] b;
  inout [15:0] c;
  …
endmodule
```

The above example declares the module ***my_module*** with three ports: ***a*** for input, ***b*** for output and ***c*** for input-output. The port ***b*** is declared of type **reg**, where the other ports are implicitly of type **wire**.

Usually after the port directions and data type declarations, nets and variables might be declared, whereas parameter declarations might be better placed even before the port directions, like the following example:

```
module my_module (a, b, c);
  parameter N = 8;
  parameter M = N * 2;
  input [N-1:0] a;
  output reg [N-1:0] b;
  inout [M-1:0] c;
  wire w1, w2;
  reg r1, r2, r3;
  …
endmodule
```

The parameter declaration might also appear in-line, before the port list inside parentheses, like the following examples, where two different methods are used:

```
module my_module #(parameter N = 8, parameter M = 16) (a, b, c);
  input [N-1:0] a;
  output reg [N-1:0] b;
  inout [M-1:0] c;
  wire w1, w2;
  reg r1, r2, r3;
  …
endmodule
```

```
module my_module #(.N(8), .M(16)) (a, b, c);
  input [N-1:0] a;
  output reg [N-1:0] b;
  inout [M-1:0] c;
  wire w1, w2;
  reg r1, r2, r3;
  …
endmodule
```

2.10 Module instance

To use a module it must be instantiated, creating a ***module instance***. For example, the following code creates the module instance ***mymo1*** from the module ***my_module*** defined at the previous section end:

```
my_module mymo1 (x1, x2, x3);
```

The module instance ***mymo1*** of the example above, connects the nets or variables ***x1***, ***x2*** and ***x3*** to the original ports ***a***, ***b*** and ***c***, following the same order. To be more clear the module instance connection might be specified, like the following example:

```
my_module mymo1 (.a(x1), .b(x2), .c(x3));
```

A module instance can change some parameter values. If the parameters are specified in-line, inside the module declaration, like the last examples of the previous section, the module instance can be obtained as the following example shows:

```
my_module #(16, 32) mymo1 (.a(x1), .b(x2), .c(x3));
```

On the above example, the instance ***mymo1*** is created assigning to the first parameter the value 16 and to the second the value 32. To be sure to select the right parameter the following variation might be used:

```
my_module #(.N(16), .M(32)) mymo1 (.a(x1), .b(x2), .c(x3));
```

If the parameters are not specified in-line in the module declaration, they can be modified after the module instance is created, with the **`defparam`** instruction:

```
my_module mymo2 (.a(x1), .b(x2), .c(x3));
defparam mymo2.DLY = 7;
```

The above example shows the instantiation of the module ***my_module*** and then, to the parameter ***DLY*** of the newly created instance is assigned the value 7.

2.11 Fully qualified path names

When the modules are designed properly, the only way to communicate with them is through ports. If, for some reason, it is necessary to access directly to an internal net or variable of an instantiated module, it is possible to use a ***fully qualified path name***. In fact, this approach might be appropriate for debugging purposes.

```
module main;
  …
  mymo3 my1 (…);
  …
endmodule
```

```
module mymo3 (…);
  …
  mymo4 my1 (…);
  …
endmodule
```

```
module mymo4 (…);
  …
  reg r1;
  …
endmodule
```

The above example shows the declaration of the top level module **`main`**, where an instance of the module **`mymo3`** is created with the name ***my1***. Then, the module **`mymo3`** contains the creation of an instance of the module **`mymo4`** with the name ***my1*** (again). The module **`mymo4`** contains the declaration of a **`reg`** type variable with the name ***r1***. To specify the variable ***r1*** of the module **`mymo4`** when instantiated in the described way, the following full path might be used:

```
main.my1.my1.r1
```

2.12 Primitives

«

The Verilog language has already available some modules describing the common logical functions and circuits: these modules are called ***primitives***. There are gate primitives that simulate a complete logic gate, and switch primitives that simulate the internal component of an electronic gate.

```
  and myand1 (y, a, b);
```

The above example shows the creation of the instance ***myand1*** that is a logic AND, receiving input from the nets or registers ***a*** and ***b***, driving output to the ***y*** connection. The instance creation might also contain a delay, for simulation purposes:

```
and #(5)   myand1 (y, a, b);
and #(5,7) myand2 (z, c, d);
```

As it can be understood, the delay is specified as a parameter value: if only one parameter is specified the delay is for rising and falling edges, otherwise the first value is the delay after the rising edge and the second for the falling one.

Gate primitives have in common the first gate argument being the output, while the other being the inputs, which usually can be one or more than one. For example, a logic AND with five inputs might be instantiated as shown by the following example:

```
and myand3 (y, a, b, c, d, e);
```

The Verilog primitives are single bit, but the instantiation can be requested for an array of the selected primitive. For example:

```
and myand4[7:0] (y, a, b);
```

In that case, the array instance ***myand4*** is created, made of eight **`and`** primitives in paralel.

Table 2.43. Gate primitives.

Primitive	Ports
`and` (***output***, ***input*** [, ***input***] ...) `or` (***output***, ***input*** [, ***input***] ...) `xor` (***output***, ***input*** [, ***input***] ...) `nand` (***output***, ***input*** [, ***input***] ...) `nor` (***output***, ***input*** [, ***input***] ...) `nxor` (***output***, ***input*** [, ***input***] ...)	Logic AND, OR, XOR, NAND, NOR and XOR.

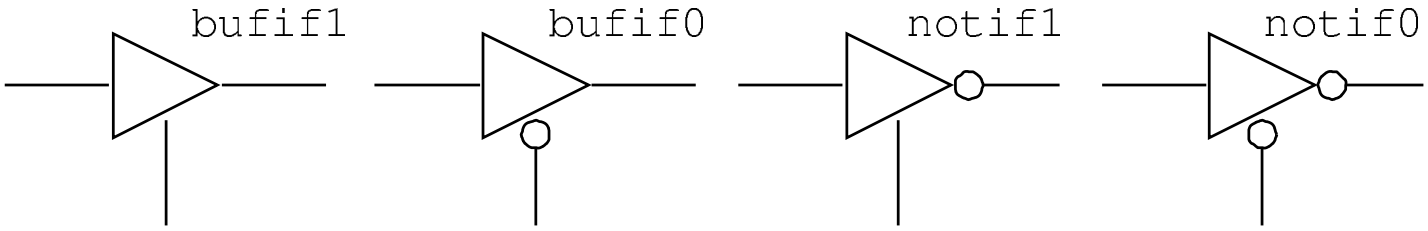
Primitive	Ports

`buf (`***output***`,` ***input***`)`
 	
`not (`***output***`,` ***input***`)`
	Buffer and inverter.

`bufif0 (`***output***`,` ***input***`,` ***control***`)`
 	
`bufif1 (`***output***`,` ***input***`,` ***control***`)`
 	
`notif0 (`***output***`,` ***input***`,` ***control***`)`
 	
`notif1 (`***output***`,` ***input***`,` ***control***`)`
	Tri-state buffers and tri-state inverters.

`pullup (`***output***`)`
 	
`pulldown (`***output***`)`
	Output equal to logic 1 or equal to logic 0.

Figure 2.44. Tri-state buffer and inverter primitives.



There are occasions when primitives can be instantiated without assigning a name to the instances, because it is not worth knowing that name. In that case the instantiations looks like a function call, but they are not functions and cannot be used inside expressions.

2.13 Expressions

«

The Verilog expressions are similar to those of the C language, even for the operator precedence and the use of parenthesis to force evaluation order. When the bit sizes of operands do not match, then the bit size of the largest value is used and every operand is extended to that size before the expression is evaluated.

Table 2.45. Verilog expression operators, grouped by precedence, starting from the highest priority, ending with the lowest one.

Operator	Description
{*a*, *b* [, *c*] ...}	Concatenation – concatenates the bits of two or more data objects (nets or registers) or expressions returning a data object. Please notice that *a* is located at the least significant position.
{*n*{*a*}}	Replication – concatenates *n* times *a*. The value *n* must be a constant.

Operator	Description
!*a*	Logic NOT – it returns 0 if *a* is not equal to zero, 1 if *a* is equal to zero.
~*a*	1's complement of *a*.
−*a*	2's complement of *a*.
&*a*	Reduction AND – if all the bits of *a* are set to 1, then returns 1, otherwise returns 0.
|*a*	Reduction OR – if all the bits of *a* are cleared to 0, then returns 0, otherwise returns 1.
^*a*	Reduction XOR – if there is an odd quantity of bits set to one, then returns 1, otherwise returns 0.
~&*a*	Reduction NAND – if all the bits of *a* are cleared to 0, then returns 1, otherwise returns 0.
~|*a*	Reduction NOR – if all the bits of *a* are set to 1, then returns 0, otherwise returns 1.
~^*a*	Reduction NXOR – if there is an odd quantity of bits set to one, then returns 0, otherwise returns 1.

Operator	Description
a ∗ *b*	Multiplication – returns the product of *a* and *b*.
a / *b*	Division – returns the quotient of *a* and *b*.

Operator	Description
a*%*b	Remainder, modulo – returns the remainder of ***a***/***b***.

Operator	Description
a*+*b	Addition – returns the sum of ***a*** and ***b***.
a*−*b	Subtraction – returns the difference of ***a*** and ***b***.

Operator	Description
a*>>*b	Logical right shift – returns the value of ***a*** shifted right ***b*** times.
a*<<*b	Logical left shift – returns the value of ***a*** shifted left ***b*** times.
a*>>>*b	Arithmetic right shift – returns the value of ***a*** shifted right ***b*** times, arithmetically.
a*<<<*b	Arithmetic left shift – returns the value of ***a*** shifted left ***b*** times, arithmetically.

Operator	Description
a*>*b	Greater then – returns 1 if ***a*** is greater than ***b***, otherwise returns 0.
a*<*b	Less then – returns 1 if ***a*** is less than ***b***, otherwise returns 0.
a*>=*b	Greater then or equal – returns 1 if ***a*** is greater than or equal to ***b***, otherwise returns 0.
a*<=*b	Less then or equal – returns 1 if ***a*** is less than or equal to ***b***, otherwise returns 0.

Operator	Description
a*==*b	Equality – returns 1 if ***a*** and ***b*** are equal and 0 if they are not. Returns unknown (`x`) if any bit inside ***a*** or ***b*** are unknown or floating (`z`).

Operator	Description
a != b	Inequality – returns 0 if a and b are equal and 1 if they are not. Returns unknown (`x`) if any bit inside a or b are unknown or floating (`z`).
a === b	Case equality – returns 1 if a and b are exactly the same, included unknown and floating bits, otherwise it returns 0.
a !== b	Case inequality – returns 0 if a and b are exactly the same, included unknown and floating bits, otherwise it returns 1.

Operator	Description
a & b	Bitwise AND – returns a AND b, bit by bit.
a ~& b	Bitwise NAND – returns $\overline{(a \text{ AND } b)}$, bit by bit.

Operator	Description
a ^ b	Bitwise XOR – returns a XOR b, bit by bit.
a ~^ b	Bitwise NXOR – returns $\overline{(a \text{ XOR } b)}$, bit by bit.

Operator	Description
a | b	Bitwise OR – returns a OR b, bit by bit.
a ~| b	Bitwise NOR – returns $\overline{(a \text{ OR } b)}$, bit by bit.

Operator	Description
a && b	Logical AND – returns 1 if a and b have both a value different from zero, otherwise it returns zero. If there are unknown or floating bits, it might be impossible to determine the result: in that case the result is unknown as well.

Operator	Description
a `||` ***b***	Logical OR – returns 1 if either ***a*** or ***b*** contains a value different from zero, otherwise it returns zero. If there are unknown or floating bits, it might be impossible to determine the result: in that case the result is unknown as well.

Operator	Description
a `?` ***b*** `:` ***c***	Conditional operator – returns ***b*** if ***a*** is different from zero, returns ***c*** if ***a*** is equal to 0; if ***a*** is unknown or floating, the result is the same as ***b*** `^` ***c*** (bitwise ***b*** XOR ***c***).

2.14 Vector and array addressing

A multi-bit net or variable (vector) can be addressed globally —as usual— or partially. To address a single bit the following syntax is used:

name [***bit***]

For example, the variable ***r1*** is declared having an 8-bit size, like this:

```
reg [7:0] r1;
```

To access the third bit ($r1_2$) the notation `r1[2]` might be used. The index that represents the selected bit might be a constant or an expression with variables or constants (care must be taken to avoid to select a bit position that does not exists). Also a range might be selected with the following syntax:

name [***msb*** `:` ***lsb***]

For example, to select the bit-range $r1_{5..2}$ the notation `r1[5:2]` should be used. The values for ***msb*** and ***lsb*** must be constant, or constant expressions. It is possible to use a variable index to select a bit range, with the following syntax:

name [***lsb*** + : ***size***]

For example, as above, to select the bit-range $r1_{5..2}$ the notation **`r1[2+:4]`** could be used, but this time, the value for ***lsb*** can be an expression with variables, whereas the ***size*** must be constant (or a constant expression).

To access an array element, the notation used is just the same as the bit selection for vectored nets or variables:

name [***element***]

For example, there might be a static memory of 256 bytes, defined like this:

```
reg [7:0] m1[255:0];
```

To access the byte at address 123 the notation **`m1[123]`** might be used. Ranges of array elements are not available, but the bits inside a selected element are reachable, like explained above. For example, to select the bit $m1[123]_3$, the notation **`m1[123][3]`** might be used.

2.15 Strings

Strings are numbers for Verilog and are represented as bit vectors, where each character is placed in a different bit octet. The following example declares a **`reg`** variable big enough to hold the string "Hello!":

```
reg [8*6-1:0] str;
…
  begin
    …
    str = "Hello!";
    …
  end
…
```

When the variable ***str*** is assigned, it contains exactly the value $48656C6C6F21_{16}$. If the receiving variable is bigger than the required space, it is padded just like a numeric value, on the left!

```
reg [8*8-1:0] str;
…
  begin
    …
    str = "Hello!";
    …
  end
…
```

In the above example, the variable ***str*** is assigned and then it contains the value $000048656C6C6F21_{16}$. The same way, assigning an empty string to a variable is just the same as assigning 0.

2.16 Functions

«

Verilog includes the ability to define functions in a similar way to the C language. There are two alternative syntaxes:

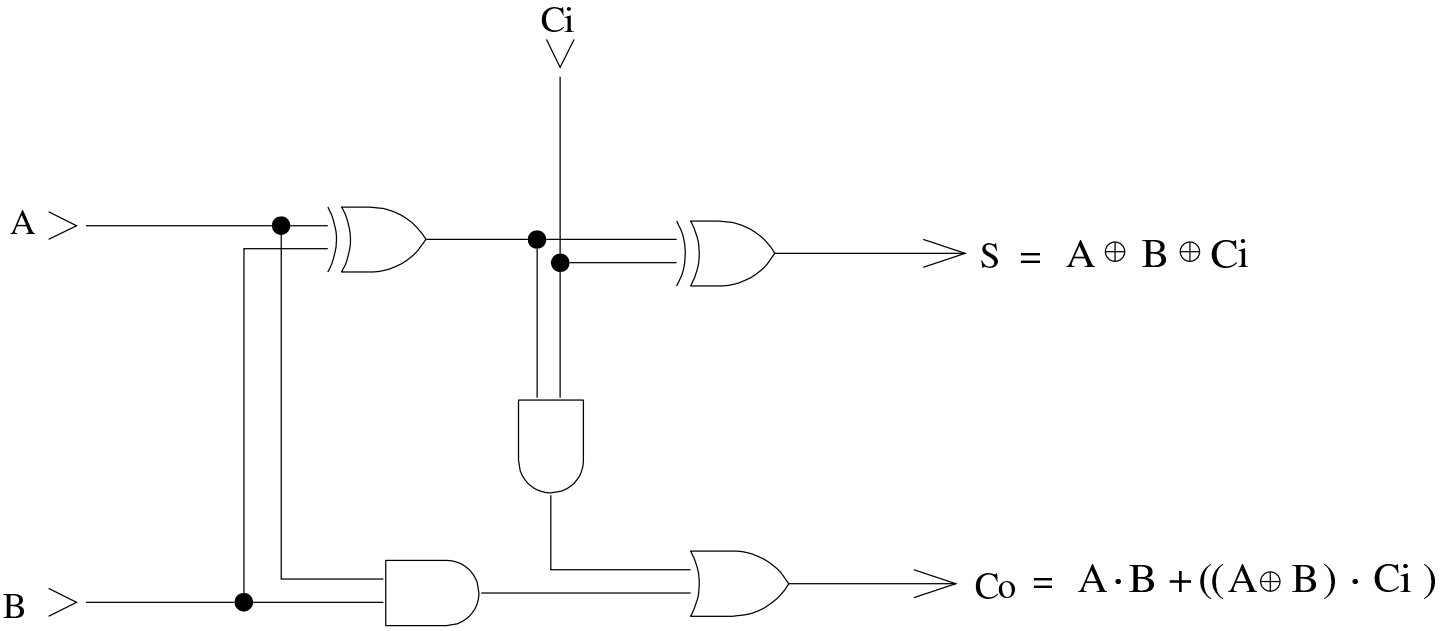
```
function [automatic] [range_or_type] function_name;
input [size] input_port;
…
begin
    statement;
    …
    function_name = value;
end
```

```
function [automatic] [range_or_type] function_name (input [size] input_port [
, …]);
begin
    statement;
    …
    function_name = value;
end
```

Verilog functions should be used in expressions, as they return a value. The value returned by a function is of the type specified by the ***range_or_type***, which is a range in the form of **[*msb*:*lsb*]** or a variable type with implicit range (like **integer**). The Verilog functions do not have a 'return' statement, instead they must assign a value to a variable with the same name as the function, at the end of it (like the syntaxes above show). The Verilog functions can have only input ports as arguments.

Normal Verilog functions are implicitly 'static', which means that the local variables are unique and shared with all concurrent calls: if two threads call the same function, it is difficult to predict the result. To make functions work like in C, where local variables are implicitly 'automatic' (because they are created inside a stack), the keyword **automatic** must be added to the function declaration. The **automatic** keyword means that each function call creates a private instance of the whole function.

Figure 2.50. Full adder.



The above figure shows the common circuit of a full adder, where for each output the logic expression is reported. The following functions reproduce the same calculations:

```
function S (input A, input B, input Ci);
begin
    S = A ^ B ^ Ci;
end
```

```
function Co (input A, input B, input Ci);
begin
    Co = A&B + ((A ^ B) & Ci);
end
```

Functions are used as a way to simplify expressions, through a sequential process that cannot contain explicit delays.

2.17 Netlist modules: combinational circuits

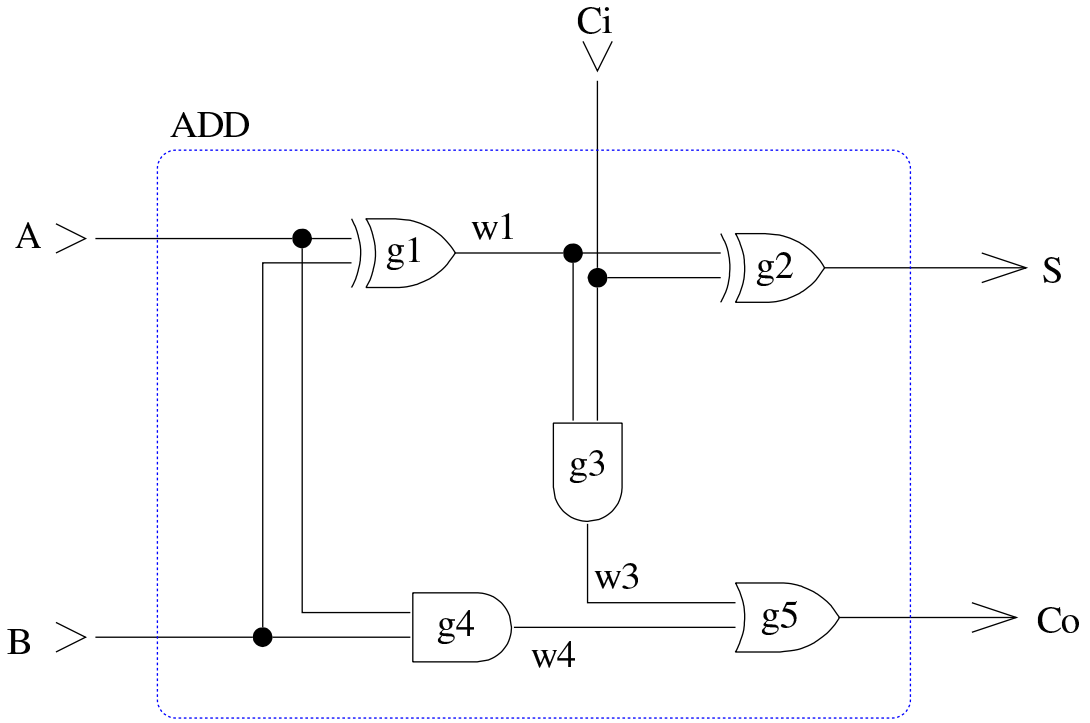
A ***netlist module*** is made of components connected only with nets, without variables. Assignments inside this kind of modules are called ***continuous***, because they just update continuously and can be defined when declaring the net or separately with the **`assign`** statement. The following examples show two alternative way for defining a continuous assignment to the net ***w1***:

```
wire w1 = a + b;
```

```
wire w1;
…
assign w1 = a + b;
```

The following figure shows a full adder netlist module that is to be written in Verilog code: all the connections and gates are named.

Figure 2.55. Full adder.



To build the above example in Verilog, as a netlist module, can be used primitives or assignments, in various ways. The following code example shows the use of primitive instantiation without assigning a particular name to the gates that they represent:

```
module ADD (S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  wire w1, w3, w4;
  xor (w1, A, B);
  xor (S, w1, Ci);
  and (w3, w1, Ci);
  and (w4, A, B);
  or (Co, w3, w4);
endmodule
```

The primitive instances can be declared with a name:

```
module ADD (S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  wire w1, w3, w4;
  xor g1 (w1, A, B);
  xor g2 (S, w1, Ci);
  and g3 (w3, w1, Ci);
  and g4 (w4, A, B);
  or g5 (Co, w3, w4);
endmodule
```

Instead of the primitives, a continuous assignment can be used:

```
module ADD (S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  wire w1, w3, w4;
  assign w1 = A ^ B;
  assign S = w1 ^ Ci;
  assign w3 = w1 & Ci;
  assign w4 = A & B;
  assign Co = w3 | w4;
endmodule
```

The continuous assignment can be defined with the wire and port declarations:

```
module ADD (S, Co, A, B, Ci);
  output S = w1 ^ Ci;
  output Co = w3 | w4;
  input A, B, Ci;
  wire w1 = A ^ B;
  wire w3 = w1 & Ci;
  wire w4 = A & B;
endmodule
```

Even functions might be used:

```
module ADD (S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  function sum (input A, input B, input Ci);
    begin
      S = A ^ B ^ Ci;
    end
  function carry (input A, input B, input Ci);
    begin
      Co = A & B + ((A ^ B) & Ci);
    end
  assign S  = sum (A, B, Ci);
  assign Co = carry (A, B, Ci);
endmodule
```

```
module ADD (S, Co, A, B, Ci);
  output S  = sum (A, B, Ci);
  output Co = carry (A, B, Ci);
  input A, B, Ci;
  function sum (input A, input B, input Ci);
    begin
      S = A ^ B ^ Ci;
    end
  function carry (input A, input B, input Ci);
    begin
      Co = A & B + ((A ^ B) & Ci);
    end
endmodule
```

The same examples can be made adding delays:

```
module ADD (S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  wire w1, w3, w4;
  xor #(8) (w1, A, B);
  xor #(8) (S, w1, Ci);
  and #(6) (w3, w1, Ci);
  and #(6) (w4, A, B);
  or #(6) (Co, w3, w4);
endmodule
```

```
module ADD (S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  wire w1, w3, w4;
  xor #(8) g1 (w1, A, B);
  xor #(8) g2 (S, w1, Ci);
  and #(6) g3 (w3, w1, Ci);
  and #(6) g4 (w4, A, B);
  or #(6) g5 (Co, w3, w4);
endmodule
```

```
module ADD (S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  wire w1, w3, w4;
  assign #(8) w1 = A ^ B;
  assign #(8) S = w1 ^ Ci;
  assign #(6) w3 = w1 & Ci;
  assign #(6) w4 = A & B;
  assign #(6) Co = w3 | w4;
endmodule
```

```
module ADD (S, Co, A, B, Ci);
  output S = #8 w1 ^ Ci;
  output Co = #6 w3 | w4;
  input A, B, Ci;
  wire w1 = #6 A ^ B;
  wire w3 = #6 w1 & Ci;
  wire w4 = #6 A & B;
endmodule
```

```
module ADD (S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  function sum (input A, input B, input Ci);
    begin
      S = A ^ B ^ Ci;
    end
  function carry (input A, input B, input Ci);
    begin
      Co = A & B + ((A ^ B) & Ci);
    end
  assign #16 S  = sum (A, B, Ci);
  assign #14 Co = carry (A, B, Ci);
endmodule
```

```
module ADD (S, Co, A, B, Ci);
  output S  = #16 sum (A, B, Ci);
  output Co = #14 carry (A, B, Ci);
  input A, B, Ci;
  function sum (input A, input B, input Ci);
    begin
      S = A ^ B ^ Ci;
    end
  function carry (input A, input B, input Ci);
    begin
      Co = A & B + ((A ^ B) & Ci);
    end
endmodule
```

2.18 Behavioral modules

The module description can contain two types of code block: **initial** or **always**. These blocks of code describe a sequence of operations, like it happens in the C language, but each block declared inside the module corresponds to a different thread. The **initial** blocks are executed only once, whereas the **always** blocks are executed repeatedly in a never ending loop. Inside a module there might be any number of **initial** and **always** blocks. The sequential code that is contained inside these blocks is delimited by the keywords **begin** and **end**, because the curly brackets are used as operators.

```
module ADD (S, Co, A, B, Ci);
  output reg S, Co;
  input A, B, Ci;
  always
    begin
      S = A + B + Ci;
      Co = ~(A ^ B ^ Ci);
    end
endmodule
```

The above example transforms the netlist examples of the previous section in the form

of a behavioral module. In this case, it is first calculated the sum of the inputs and then the carry out, but as there are no delays, it all happens simultaneously. Please notice that the output ports are now changed into a **reg** type, because inside the behavioral blocks the assignments need to modify a **reg** type destination.

2.19 Procedural assignments

The assignments inside the procedural blocks (behavioral) are known as ***procedural assignments***, as they differ from the continuous assignments used outside blocks. The procedural assignments require to have a **reg** type variable on the left side of the assignment, whereas the continuous assignments require a net type destination.

There are two types of procedural assignments: blocking and non-blocking. ***Blocking assignments*** appear and work like the C language assignments. The adjective 'blocking' means that the assignment must be done before the next sentences are executed.

Inside the following example —which already appeared in a previous section— there are two blocking assignments: they just mean that first the sum is assigned to the output variable ***S*** and then that the carry out is assigned to the output variable ***Co***.

```
module ADD (S, Co, A, B, Ci);
  output reg S, Co;
  input A, B, Ci;
  always
    begin
      S = A + B + Ci;
      Co = ~(A ^ B ^ Ci);
    end
endmodule
```

Non-blocking assignments are similar to the blocking ones, but they are executed in a separate thread. The following example does the same work as the previous one, but the assignments just happens simultaneously:

```
module ADD (S, Co, A, B, Ci);
  output reg S, Co;
  input A, B, Ci;
  always
    begin
      S <= A + B + Ci;
      Co <= ~(A ^ B ^ Ci);
    end
endmodule
```

Assignments can be delayed, using the following syntax, for blocking and non-blocking assignments:

dst = [#***delay***] ***expression***

dst <= [#***delay***] ***expression***

The following example is a modified version of the blocking assignment one, shown above, adding a 14 time units delay before the sum is calculated:

```
module ADD (S, Co, A, B, Ci);
  output reg S, Co;
  input A, B, Ci;
  always
    begin
      S = #14 A + B + Ci;
      Co = ~(A ^ B ^ Ci);
    end
endmodule
```

In the above example, the value for ***Co*** is calculated only after the assignment for ***S***, so the output variables are updated at the same time, with the same delay. The following example uses non-blocking assignments and it has a different behaviour,

because the two assignments are independent each other:

```
module ADD (S, Co, A, B, Ci);
  output reg S, Co;
  input A, B, Ci;
  always
    begin
      S <= #14 A + B + Ci;
      Co <= #12 ~(A ^ B ^ Ci);
    end
endmodule
```

In the above example, the second assignment does not wait for the first one and the delay is less than the first one. This means that the ***Co*** output variable is update before the ***S*** variable.

2.20 Blocking delays

The execution of each statement inside a procedural block can be delayed, placing a delay number before the statement itself:

#delay [***procedural_statement***] ;

Please notice that the delayed statement is optional, so that the delay can be alone, like an autonomous statement. The following example shows a variation of the delayed blocking assignments, with the delay specified before the assignment statement:

```
module ADD (S, Co, A, B, Ci);
  output reg S, Co;
  input A, B, Ci;
  always
    begin
      #14 S =  A + B + Ci;
      Co = ~(A ^ B ^ Ci);
    end
endmodule
```

Please notice that

```
#14 S =  A + B + Ci;
```

is just the same as

```
#14;
S =  A + B + Ci;
```

When non-blocking assignments are used, care must be taken using the blocking assignments; for example,

```
S <= #14 A + B + Ci;
Co <= #12 ~(A ^ B ^ Ci);
```

is much different from the following:

```
#14 S <=  A + B + Ci;
#12 Co <= ~(A ^ B ^ Ci);
```

The last example above just means: wait 14 time units, then assign the sum to ***S*** (without waiting), then wait another 12 time units, then assign the carry out to ***Co*** (without waiting). In fact, the carry out is assigned after 26 time unit.

2.21 Wait for a level event

«

Inside a procedural block it is possible to wait for a value being available in a net or variable data type: this is called ***level event***. The **`wait`** statement is used to test a level event:

`wait` (***expression***) [***statement***]

```
wait (expression)
   [begin
      statement
      …
   end]
```

The following example should implement a D latch:

```
module latch_d (Q, _Q, D, Clk);
  input D, Clk;
  output reg Q, _Q;
  always
    begin
      wait (Clk == 1)
        begin
          #10 Q = D;
          _Q = ~D;
        end
    end
endmodule
```

The above example just waits for the ***Clk*** input to be 1, then it updates the outputs (with a 10 time units delay); if the ***Clk*** is already at the 1 level, the outputs keep updating. The statement (or the group of statements) that follows the wait condition, might be omitted, if there is only the need to wait for the level event:

```
module latch_d (Q, _Q, D, Clk);
  input D, Clk;
  output reg Q, _Q;
  always
    begin
      wait (Clk == 1);
      #10 Q = D;
      _Q = ~D;
    end
endmodule
```

2.22 Wait for an event expressions

«

The following syntax is used to wait until an event condition is true (triggered). Please notice that the event condition is different from the level condition used by the **`wait`** statement.

@ (***event_expression***) [***statement***]

@ (***event_expression***)
[begin
statement
...
end]

The event expression is different from common expressions, because it is meant to be true when a specified event occur.

Table 2.80. Basic event expressions.

Expression	Description
a	When the ***a*** net or variable changes value, an event occurs.
`posedge` ***a***	When the ***a*** net or variable changes value becoming 1 (positive edge).
`negedge` ***a***	When the ***a*** net or variable changes value becoming 0 (negative edge).
e1 `or` ***e2***	When any of the two events (***e1*** or ***e2***) becomes true.

The following example should implement a D flip-flop triggered by a positive edge clock signal. Please notice that the **`always`** statement is followed by the `@` statement, which owns the **`begin-end`** block.

```
module ff_d (Q, _Q, D, Clk);
  input D, Clk;
  output reg Q, _Q;
  always @ (posedge Clk)
    begin
      Q = #10 D;
      _Q = ~Q;
    end
endmodule
```

The same thing could also be done using the `@` statement just to wait time, blocking the procedure flow until the event occurs:

```
module ff_d (Q, _Q, D, Clk);
  input D, Clk;
  output reg Q, _Q;
  always
    begin
      @ (posedge Clk);
      Q = #10 D;
      _Q = ~Q;
    end
endmodule
```

2.23 Event variables

Event variables are only able to store the triggering of an event, with the **->** operator:

```
begin
  …
  -> e;
  …
end
```

An event can be risen inside an event variable only in a procedural context, as the above example might suggest. An event variable can be used as an event expression or subexpression:

```
@ (e) …;
```

2.24 System tasks and functions

Verilog provides some special functions for the simulation purpose, to be used inside procedural blocks, characterized by the **$** (dollar) prefix, useful mainly for debugging. The most important of these functions is **$display()**, which allows to show some text (to the simulation output console) in a similar way to **printf()** for the C language.

```
module …;
  …
  reg [7:0] x, y, z;
  …
  initial
    begin
      w = 8'h41;          // A
      x = 8'hab;
      y = 8'hcd;
      z = 8'hef;
      $display ("Hello: w = %c, x = %d, y = 0x%h, z = %b",
                w, x, y, z);
    end
  …
endmodule
```

The above example should display the following text on the simulation console:

```
Hello: w = A, x = 171, y = 0xcd, z = 11101111
```

Table 2.87. Common system functions.

System function	Description
`$display (`***string*** `[, ` ***x*** `] …)`	It displays on the simulation console the string, expanding the metavariables `%…` with the value provided by the nets and variables that forms the following arguments. It works like the **`printf()`** function for the C language, except that it is displayed with a new-line at the end.

System function	Description
`$monitor (`***string*** `,` ***x*** `[,` ***y*** `]` ...`)`	Continuous monitoring: whenever one of the arguments following the string changes value, the string is displayed on the simulation console, expanding the metavariables `%`... with the value provided by the nets and variables that forms the following arguments. It works like the **`printf()`** function for the C language, except that it is displayed with a new-line at the end. Generally, this function is used only on **`initial`** blocks, because once requested, it remains active.
`$time`	It returns a value corresponding to the current time, which is expressed in the unit specified with the **`` `timescale``** directive. The time value can be assigned to a **`time`** variable type and can be shown inside **`$display()`** or similar functions with the metavariable **`%t`**.
`$finish`	It stops the simulation.

2.25 Control structures

«

Inside a procedural block, many of the common C language control structures can be used. However it must be recalled that the group of statements is made in Verilog language with the keywords **`begin-end`**, because the curly brackets are used as operators.

Syntax 2.88. Conditional statement: **`if`**.

```
if (condition)
     statement ;
[else
     statement ; ]
```

Syntax 2.89. Selection statement: **`case`**.

```
case | casez | casex (expression)
   case_expression :  statement ;
   ...
   [default: statement ; ]
endcase
```

The selection statement requires some explanations. The first thing that should be noticed is that the value in parentheses —that should be compared with the case list below— is an expression, which is evaluated at run time. The same is for the case conditions appearing below, which are expressions evaluated at run time. Then, after each case condition there is only one statement and there is no ‘break‘ keyword (the single statement might be expanded with the keywords **`begin-end`**).

There are three type of case statements: **`case`** matches the expression in parentheses exactly, even the floating and unknown values (**`z`** and **`x`**) should be the same; **`casez`** considers that floating values are ***don’t-care*** values (a don’t-care bit value always match); **`casex`** considers that floating and unknown are don’t-care values.

Syntax 2.90. Loop statement: **`while`**.

```
while (condition)
    statement ;
```

Syntax 2.91. Loop statement: **`for`**.

```
for (initialization ; condition ; statement)
    statement ;
```

The following example shows a typical use of the **`for`** loop; please notice that the ***i*** variable is declared as **`integer`** type:

```
module …;
  …
  reg [7:0] m[1023:0];
  integer i;
  …
  always
    begin
      …
      for (i = 0; i < 1024; i = i + 1)
        begin
            …
        end
      …
    end
    …
endmodule
```

Syntax 2.93. Loop statement: **repeat**.

```
repeat (count)
    statement ;
```

The **repeat** loop executes the statement (or the group of statements) a fixed number of time, as defined by the ***count*** expression. If the ***count*** expression has an invalid value, it is treated as zero.

Syntax 2.94. Loop statement: **forever**.

```
forever
    statement ;
```

The **forever** loop executes the statement (or the group of statements) indefinitely and the statements after the structure would never be executed.

2.26 Thread control

Every procedure inside an **initial** or **always** block has a different thread, but inside a procedure a group of threads that should be synchronized can be declared with a special block called **fork-join**.

```
fork
    statement ;
    statement ;
    ...
join
```

The statements inside the **fork-join** block are started simultaneously, but before leaving the block it is ensured that all these statements have completed their work.

If instead of a single statement a **`begin-end`** group is placed, the contained statements are executed sequentially in a single thread.

[1] Identifiers might also contain other escaped characters, but this possibility should be avoided and it is not taken into account inside the chapter.

[2] Please notice that the syntax is simplified and it does not consider the signedness, because the chapter takes into account only the unsigned representation.

[3] The value 232_{10} is equal to 11101000_2, but allowing only seven bits, the value is truncated to 1101000_2 that is equal to 104_{10}.

Combinational circuits

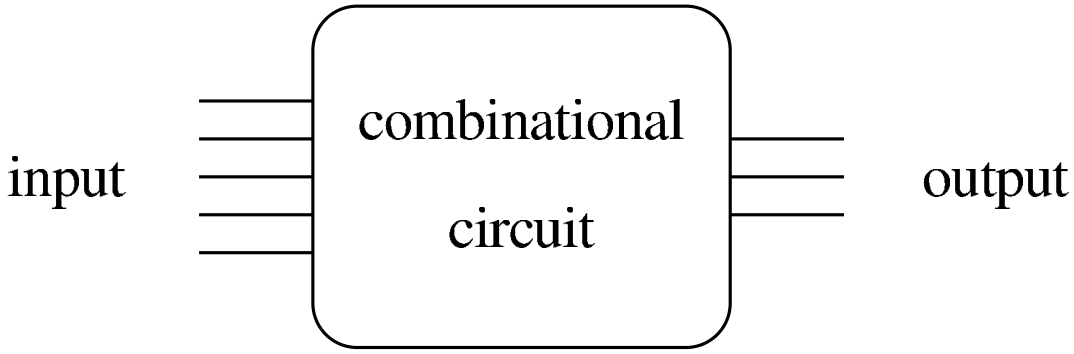
3.1 Decoder 87
3.2 Demultiplexer 90
3.3 Multiplexer 93
3.4 Multiplexer and demultiplexer with paralel I/O 96
3.5 Binary encoder 99
3.6 Priority encoder 102
3.7 Logic units 106
3.8 Shift and rotation 107
3.9 Addition 114
3.10 Subtraction 119
3.11 Addition and subtraction together 123
3.12 Carry lookahead 125
3.13 2's complement 132
3.14 Multiplication 133
3.15 Division 142
3.16 Magnitude comparator 145

2's complement 132 adder 114 arithmetic shift 107 binary encoder 99 carry lookahead 125 combinational circuit 83 combinational network 83 data distributor 90 data selector 93 decoder 87 demultiplexer 90 demux 90 encoder 99 fast adder 125 full-adder 114 half-adder 114 half-subtractor 119 logic shift 107 logic unit 106 multiplexer 93 priority encoder 102 ripple-carry 114 rotation 107 shift 107 subtraction 119 two's complement 132

A ***combinational circuit***, otherwise known as ***combinational network***, is a system of logic gates conveniently connected together, organized with a set of input ports and a set of output ports, where the output logic values are directly and univocally defined

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

from the input logic ones. The overall combinational circuit can be represented as a box with input and output ports, together with a truth table that describes the output values based on the input combinations.

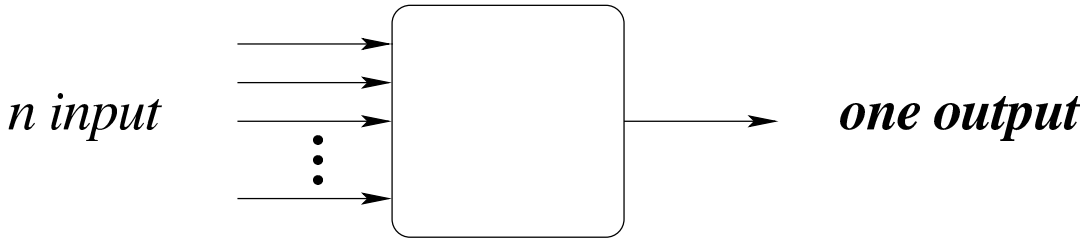


The figure above shows a combinational circuit example with five input ports and three output ports, but the proportion between input and output ports depends on the function that should be performed by the circuit, or rather from the purpose that the circuit should satisfy.

Please notice that a combinational circuit is not influenced by the time variable and even by the power-on random variable; therefore, inside these circuits, the propagation delay is a problem that is not taken much into consideration, because, after a certain delay from the input change, the output is updated as specified by the truth table.

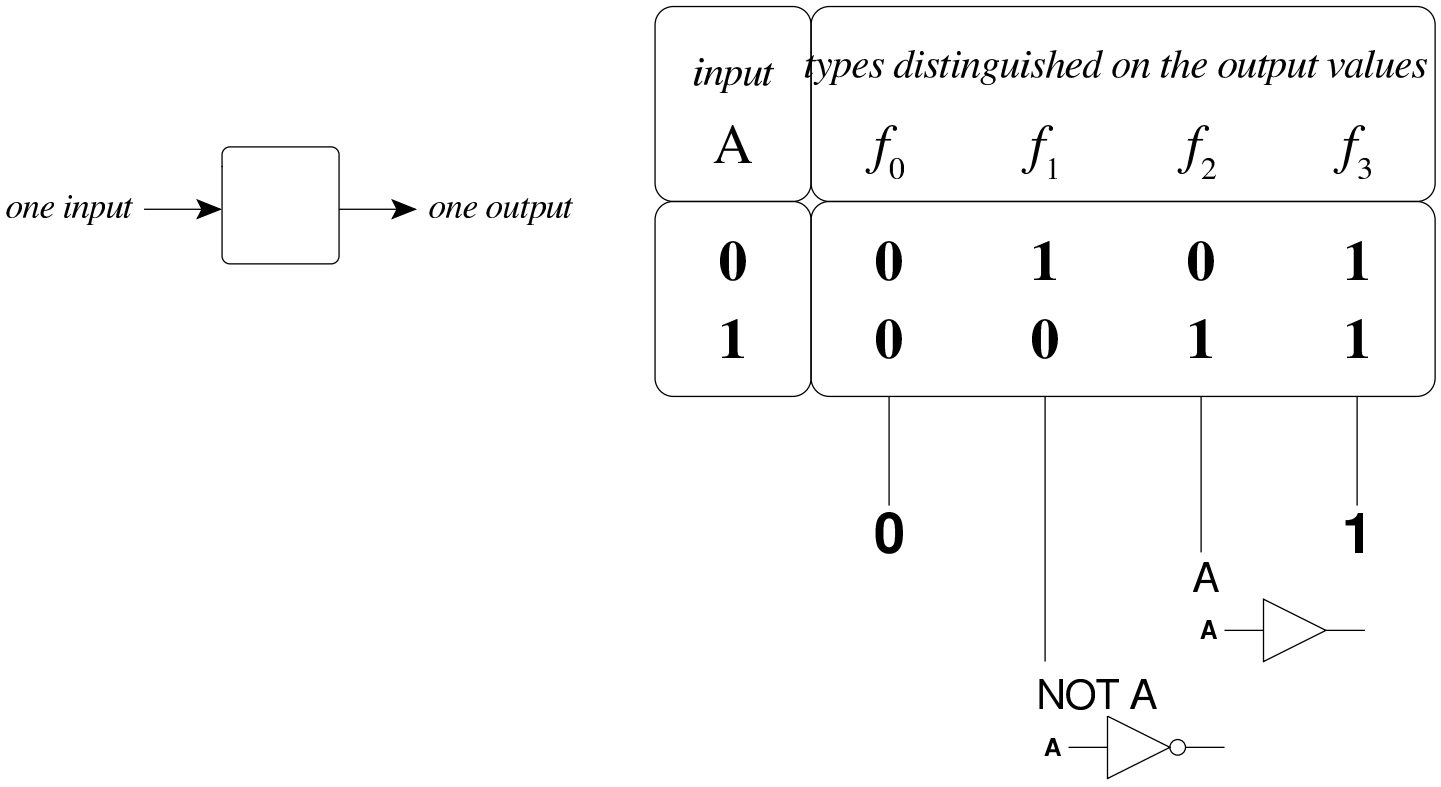
The simpler combinational circuits are those that have only one output port, which are better known as ***logic gates***.

Figure 3.2. Combinational circuits with a single output port.



To understand better the matter, it can be noticed that a circuit composed only by a single input port and a single output port can be made in four different ways, as evidenced by the following figure.

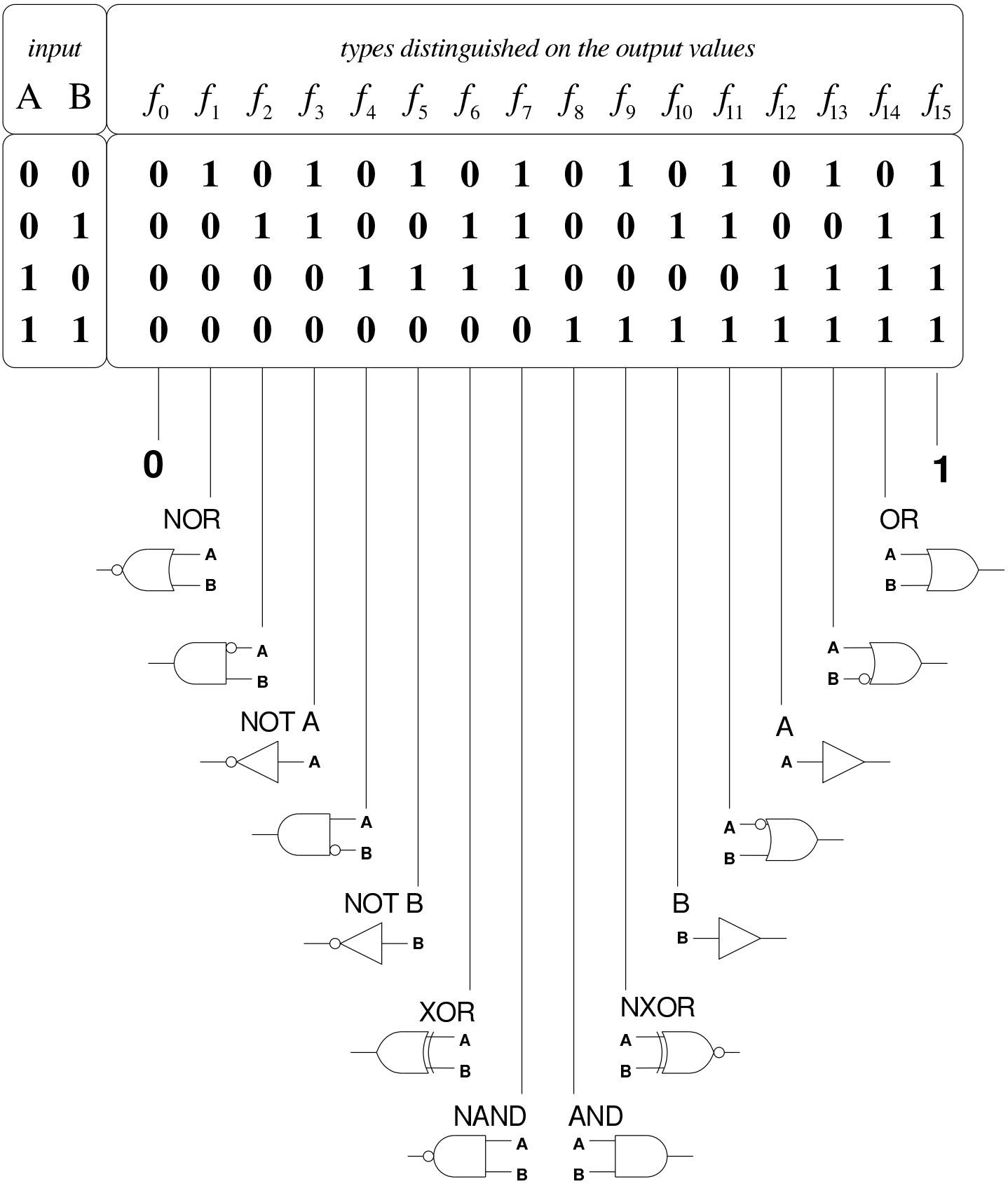
Figure 3.3. Truth table about four functions of a single variable.



As it can be seen on the annotations inside the above figure, the circuit corresponding to the f_1 function is the inverter (NOT), whereas the one corresponding to the f_2 function is the buffer (not-inverter).

A combinational circuit with **two input ports** and a single output can have 16 alternative functions, as it can be seen in the following figure, where the functions corresponding to the common logic ports are evidenced.

Figure 3.4. Truth table about sixteen functions of two variables.



3.1 Decoder

The ***decoder*** is a combinational circuit that, for every input combination, asserts only one output port. Precisely, for every input combination there is only one output port to assert. Therefore, for n input ports there are 2^n output ports. Usually, this kind of combinational circuit has also an additional input control port that should be asserted to enable the output.

Figure 3.5. Block diagram of a decoder with four output ports (2-to-4). The two input selection ports are labeled as A_0 and A_1 (address); the input control port is labeled as E (enable); the output ports are labeled from Y_0 to Y_3. The drawing on the right is more compact and the input selection ports are grouped together.

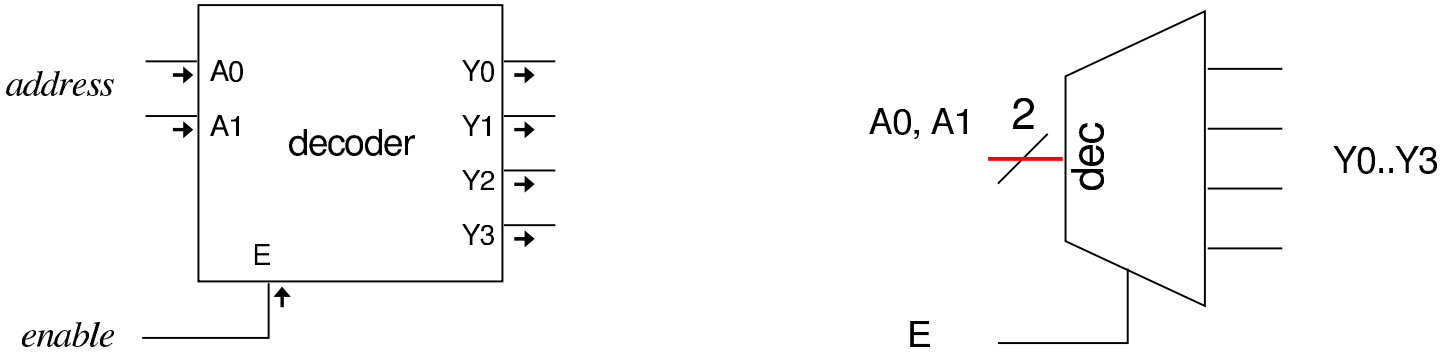
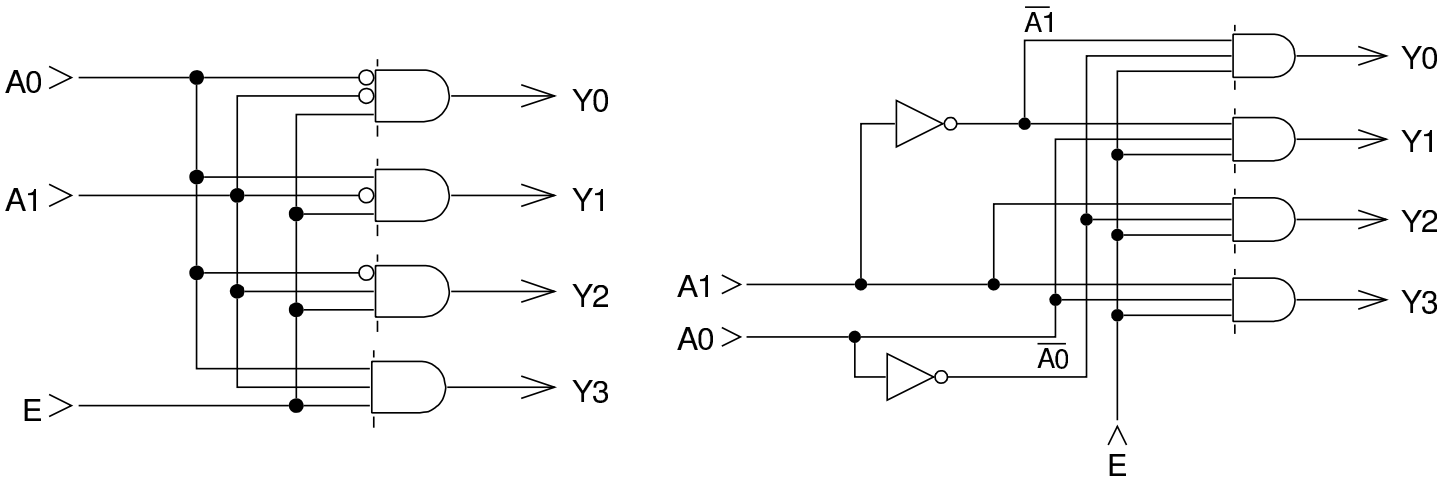


Table 3.6. Truth table for a decoder with four output ports, from Y_0 to Y_3; two selection input ports, A_0 and A_1; one enable input port E.

E	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Figure 3.7. Two examples of implementation of a decoder with four output ports: the two selection ports are labeled A_0 and A_1, the enable port is labeled E and the output ports are labeled from Y_0 to Y_3.



Listing 3.8. Example with Verilog.

```
module decoder_2_to_4 (Y, A, E);
  input [1:0] A;
  input E;
  output [3:0] Y;
  //
  function [3:0] f2to4 (input [1:0] A, input E);
    if (E == 1)
      begin
        case (A)
          0: f2to4 = 4'b0001;
          1: f2to4 = 4'b0010;
          2: f2to4 = 4'b0100;
          3: f2to4 = 4'b1000;
        endcase
      end
    else
      begin
        f2to4 = 4'b0000;
      end
  endfunction
```

```
  //
  assign #8 Y = f2to4 (A, E);
endmodule
```

Listing 3.9. Verilog gate level alternative code following the picture 3.7 on the left.

```
module decoder_2_to_4 (Y, A, E);
  input [1:0] A;
  input E;
  output [3:0] Y;
  assign Y[0] = E & ~A[1] & ~A[0];
  assign Y[1] = E & ~A[1] &  A[0];
  assign Y[2] = E &  A[1] & ~A[0];
  assign Y[3] = E &  A[1] &  A[0];
endmodule
```

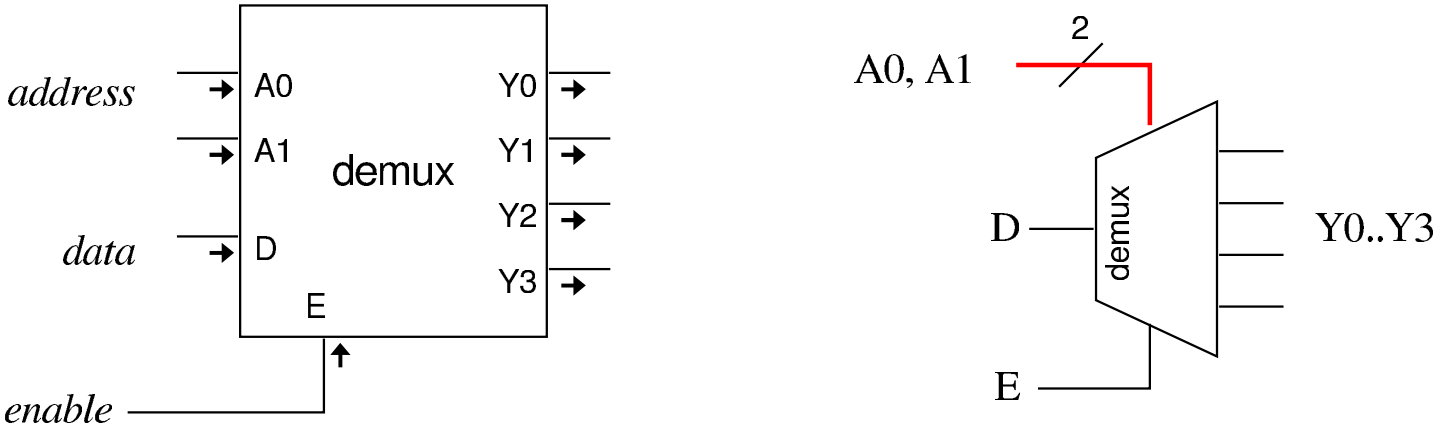
Listing 3.10. Verilog gate level alternative code following the picture 3.7 on the right.

```
module decoder_2_to_4 (Y, A, E);
  input [1:0] A;
  input E;
  output [3:0] Y;
  wire [1:0] _A;
  assign _A = ~A;
  assign Y[0] = E & _A[1] & _A[0];
  assign Y[1] = E & _A[1] &  A[0];
  assign Y[2] = E &  A[1] & _A[0];
  assign Y[3] = E &  A[1] &  A[0];
endmodule
```

3.2 Demultiplexer

The ***demultiplexer*** (demux), known also as ***data distributor***, is a combinational circuit that works like a decoder with an additional input port switched to the selected output.

Figure 3.11. Block diagram of a demultiplexer with four output ports. The two selection input ports are labeled A_0 and A_1, the data input corresponds to the variable D, the enable input is E, the output ports are labeled from Y_0 to Y_3. The drawing on the right is more compact and the input selection ports are grouped together.

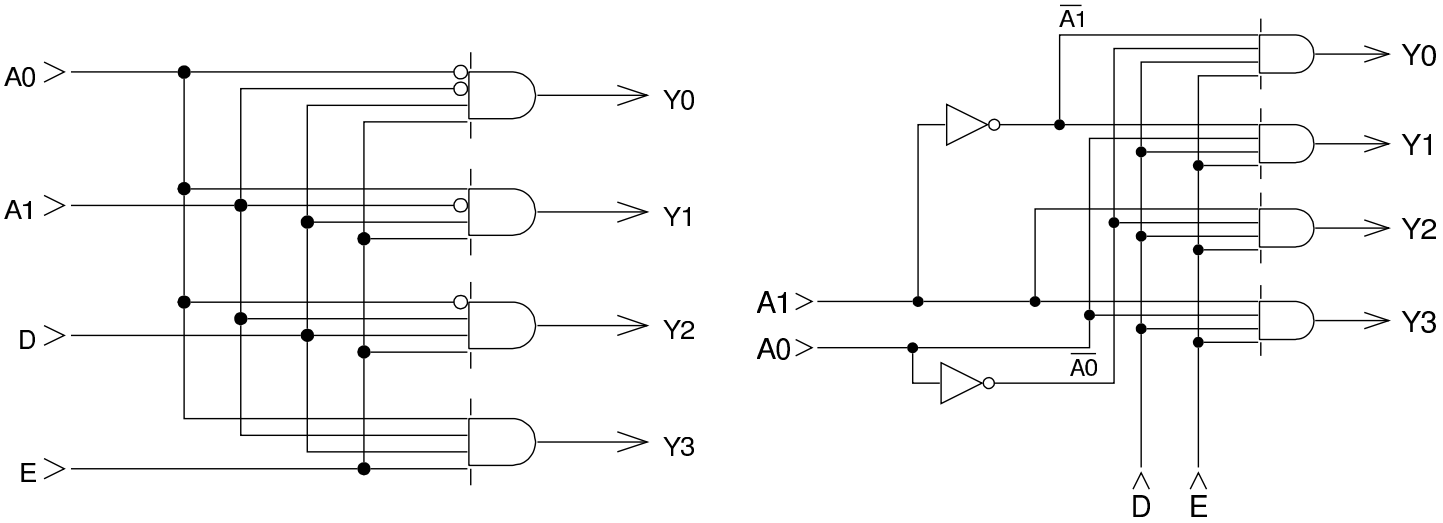


The difference between the decoder and the demultiplexer should be clear from the figure above: the Y_n output port selected has the same value read at the D input (unless the enable port is negated).

Table 3.12. Truth table for a demultiplexer with four output ports, from Y_0 to Y_3; two selection input ports, A_0 and A_1; a data input S; an enable input E.

E	A_1	A_0	D	Y_3	Y_2	Y_1	Y_0
0	X	X	***d***	0	0	0	0
1	0	0	***d***	0	0	0	***d***
1	0	1	***d***	0	0	***d***	0
1	1	0	***d***	0	***d***	0	0
1	1	1	***d***	***d***	0	0	0

Figure 3.13. Two examples of implementation of a demultiplexer with four output ports: the two selection ports are labeled A_0 and A_1, the data to be switched to the selected output is read from the input D, the enable port is labeled E and the output ports are labeled from Y_0 to Y_3.



It should be noticed that D and E work just like alternative enabling inputs, so the truth table might be simplified as it appears below.

Table 3.14. Alternative way to represent the truth table of the demultiplexer with four outputs. Please notice that here the data and enable inputs are both at the beginning.

E	D	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	X	X	X	0	0	0	0
X	0	X	X	0	0	0	0
1	1	0	0	0	0	0	1
1	1	0	1	0	0	1	0
1	1	1	0	0	1	0	0
1	1	1	1	1	0	0	0

Listing 3.15. Example with Verilog.

```
module demultiplexer_2_to_4 (Y, A, D, E);
  input [1:0] A;
  input D, E;
  output [3:0] Y;
  //
```

```
  function [3:0] f2to4 (input [1:0] A, input D, input E);
    if (E == 1 && D == 1)
      begin
        case (A)
          0: f2to4 = 4'b0001;
          1: f2to4 = 4'b0010;
          2: f2to4 = 4'b0100;
          3: f2to4 = 4'b1000;
        endcase
      end
    else
      begin
        f2to4 = 4'b0000;
      end
  endfunction
  //
  assign #8 Y = f2to4 (A, D, E);
endmodule
```

Listing 3.16. Verilog gate level alternative code following the picture 3.13 on the left.

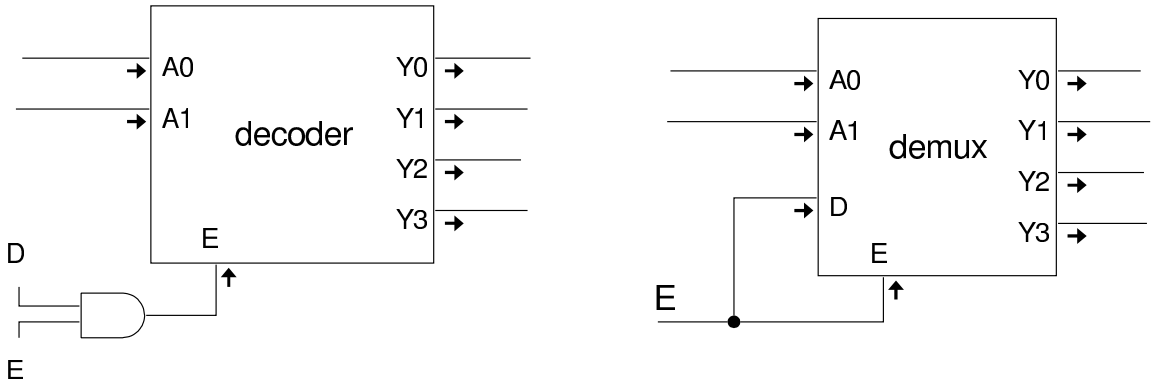
```
module demultiplexer_2_to_4 (Y, A, D, E);
  input [1:0] A;
  input E, D;
  output [3:0] Y;
  assign Y[0] = E & D & ~A[1] & ~A[0];
  assign Y[1] = E & D & ~A[1] &  A[0];
  assign Y[2] = E & D &  A[1] & ~A[0];
  assign Y[3] = E & D &  A[1] &  A[0];
endmodule
```

Listing 3.17. Verilog gate level alternative code following the picture 3.7 on the right.

```
module demultiplexer_2_to_4 (Y, A, D, E);
  input [1:0] A;
  input E, D;
  output [3:0] Y;
  wire [1:0] _A;
  assign _A = ~A;
  assign Y[0] = E & D & _A[1] & _A[0];
  assign Y[1] = E & D & _A[1] &  A[0];
  assign Y[2] = E & D &  A[1] & _A[0];
  assign Y[3] = E & D &  A[1] &  A[0];
endmodule
```

A demultiplexer can be obtained from a decoder (provided that it has the enable input port), likewise a demultiplexer can be reduced to work as a decoder.

Figure 3.18. On the left a decoder is adapted to work as a demultiplexer; on the right a demultiplexer is adapted to work as a decoder.



Common demultiplexer integrated circuits are 74154, 74138, 74139 and 74238 (see section u0.1).

3.3 Multiplexer

The ***multiplexer*** (mux), or ***data selector***, is a combinational circuit that selects the value of a single input port and reproduces that value to the output port. The input port chosen depends on the value of a group of selection ports (or address ports). For n selection ports there can be 2^n input ports.

Figure 3.19. Block diagram of a multiplexer with four input ports. The two selection inputs are labeled A_0 and A_1, the data input ports are labeled from D_0 to D_3, the enable input is E, the output is Y. On the right it is shown a compact representation where the selection variables are joined together in a multiple line.

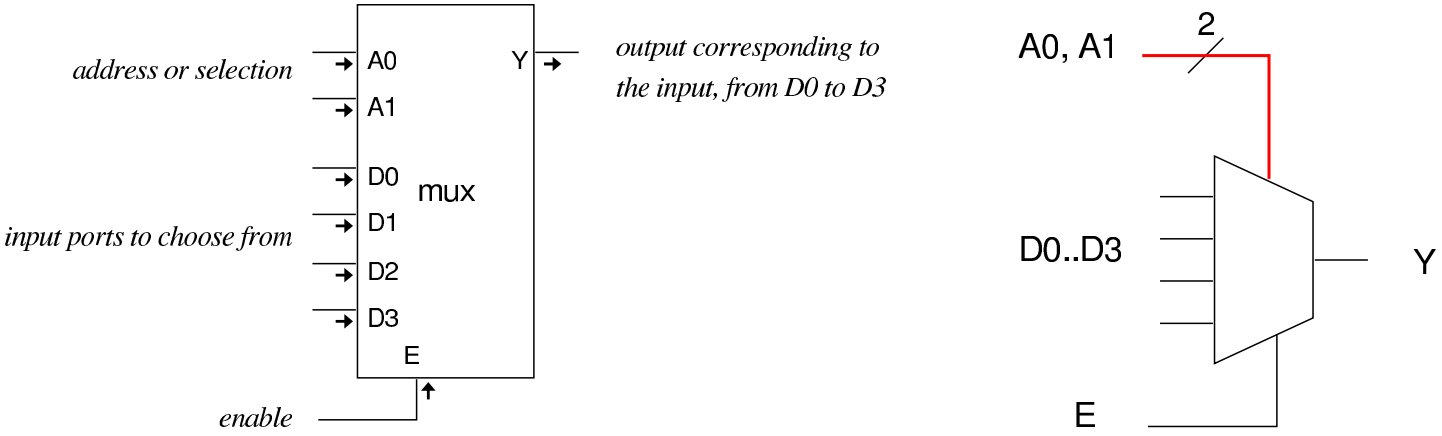
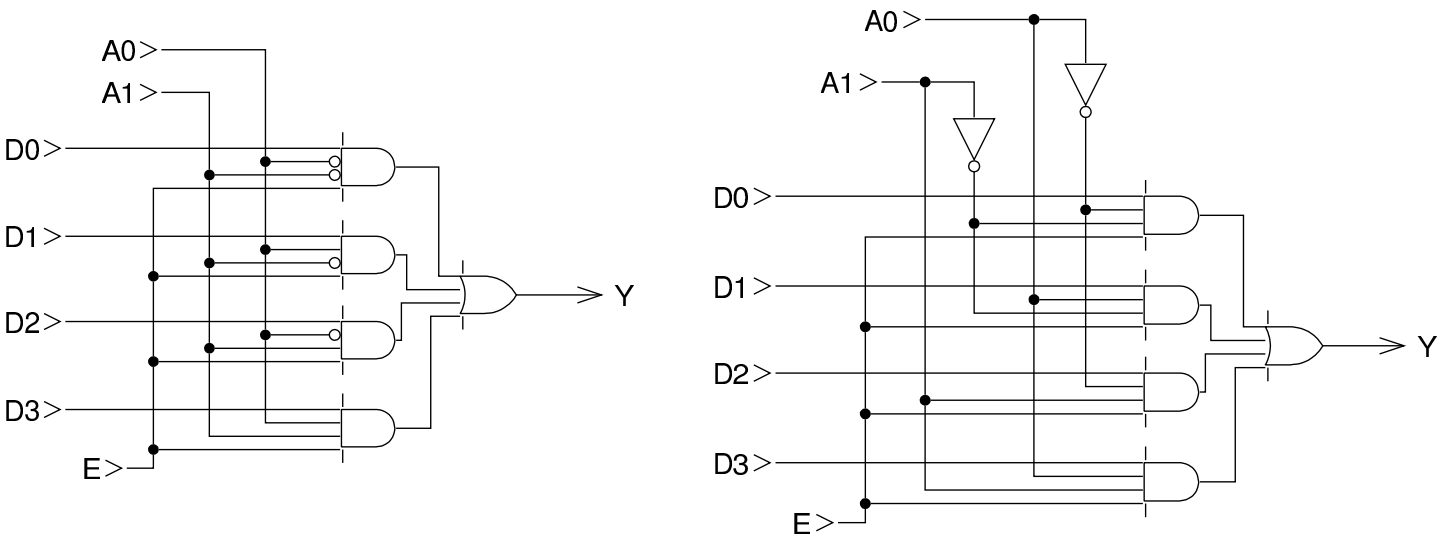


Table 3.20. Truth table for a multiplexer with four data inputs, from D_0 to D_3; two selection inputs, from A_0 to A_1; an enable input, E; one output, Y.

E	A_1	A_0	D_3	D_2	D_1	D_0	Y
0	x	x	d_3	d_2	d_1	d_0	0
1	0	0	d_3	d_2	d_1	d_0	d_0
1	0	1	d_3	d_2	d_1	d_0	d_1
1	1	0	d_3	d_2	d_1	d_0	d_2
1	1	1	d_3	d_2	d_1	d_0	d_3

Figure 3.21. Two example of a multiplexer with four data inputs.



Listing 3.22. Example with Verilog.

```
module multiplexer_1_of_4 (Y, A, D, E);
  input [1:0] A;
  input [3:0] D;
  input E;
  output Y;
  //
  function f1of4 (input [1:0] A, input [3:0] D, input E);
    if (E == 1)
      begin
        case (A)
          0: f1of4 = D[0];
          1: f1of4 = D[1];
          2: f1of4 = D[2];
          3: f1of4 = D[3];
        endcase
      end
    else
      begin
        f1of4 = 0;
      end
  endfunction
  //
  assign #8 Y = f1of4 (A, D, E);
endmodule
```

Listing 3.23. Verilog gate level alternative code following the picture 3.21 on the left.

```
module multiplexer_1_of_4 (Y, A, D, E);
  input [1:0] A;
  input [3:0] D;
  input E;
  output Y;
  wire w0, w1, w2, w3;
```

```
  assign #4 w0 = E & ~A[0] & ~A[1] & D[0];
  assign #4 w1 = E &  A[0] & ~A[1] & D[1];
  assign #4 w2 = E & ~A[0] &  A[1] & D[2];
  assign #4 w3 = E &  A[0] &  A[1] & D[3];
  assign #4 Y  = w0 | w1 | w2 | w3;
endmodule
```

Listing 3.24. Verilog gate level alternative code following the picture 3.21 on the right.

```
module multiplexer_1_of_4 (Y, A, D, E);
  input [1:0] A;
  input [3:0] D;
  input E;
  output Y;
  wire [1:0] _A;
  wire w0, w1, w2, w3;
  assign _A = ~A;
  assign #4 w0 = E & _A[0] & _A[1] & D[0];
  assign #4 w1 = E &  A[0] & _A[1] & D[1];
  assign #4 w2 = E & _A[0] &  A[1] & D[2];
  assign #4 w3 = E &  A[0] &  A[1] & D[3];
  assign #4 Y  = w0 | w1 | w2 | w3;
endmodule
```

Common multiplexer integrated circuits are 74150, 74151 and 74157 (see section u0.2).

3.4 Multiplexer and demultiplexer with paralel I/O

«

Multiplexers and demultiplexer might be required to work on data composed of more than a single bit. In that case, many multiplexers or demultiplexers are required, but joining the selection and enabling ports. The following figures show a demultiplexer and a multiplexer in a detailed version (on the left) and in a compact version with some wires grouped together (on the right).

Figure 3.25. Demultiplexer with input and outputs composed of four value vectors.

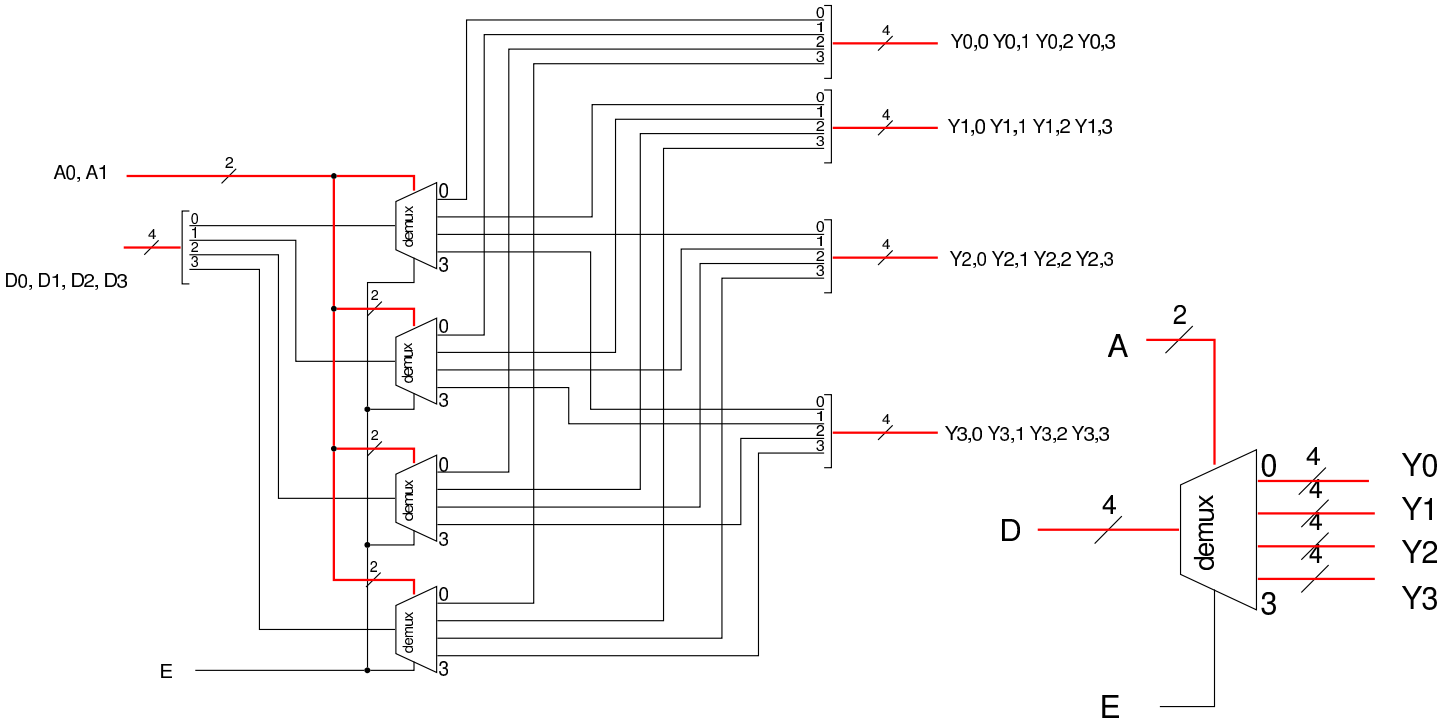
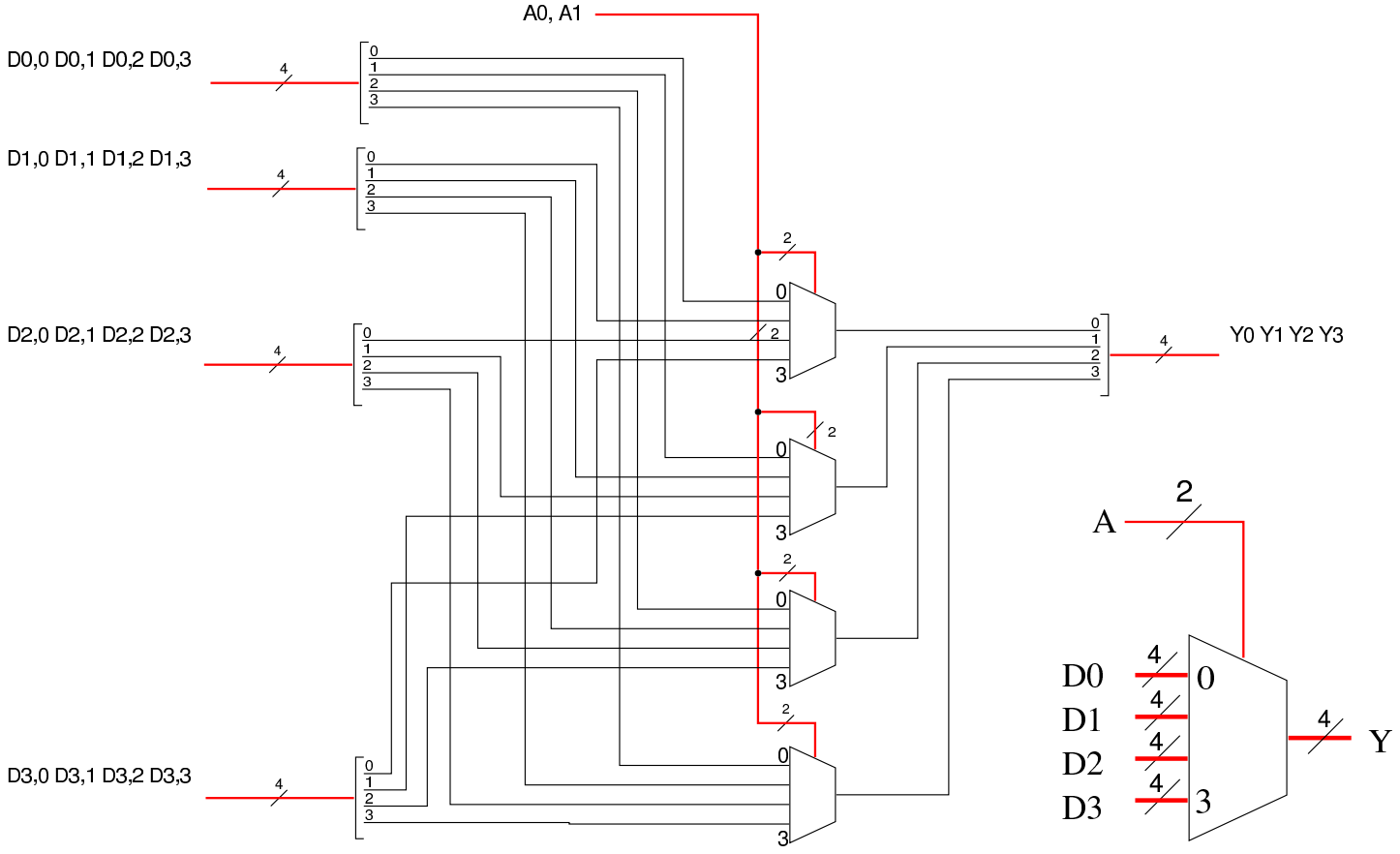


Figure 3.26. Multiplexer with inputs and output composed of four value vectors.



Listing 3.27. Quad 2-to-4 demultiplexer example in Verilog.

```
module demultiplexer_2_to_4_x4 (Y0, Y1, Y2, Y3, A, D, E);
  input [1:0] A;
  input E;
  input [3:0] D;
  output [3:0] Y0, Y1, Y2, Y3;
  wire [1:0] _A;
  assign _A = ~A;
  assign #8 Y0 = (E & _A[1] & _A[0])?D:0;
  assign #8 Y1 = (E & _A[1] &  A[0])?D:0;
  assign #8 Y2 = (E &  A[1] & _A[0])?D:0;
  assign #8 Y3 = (E &  A[1] &  A[0])?D:0;
endmodule
```

Listing 3.28. Quad 1-of-4 multiplexer example in Verilog.

```
module multiplexer_1_of_4_x4 (Y, A, D0, D1, D2, D3, E);
  input [1:0] A;
  input [3:0] D0, D1, D2, D3;
  input E;
  output [3:0] Y;
  wire [1:0] _A;
  wire [3:0] w0, w1, w2, w3;
  assign _A = ~A;
  assign #4 w0 = (E & _A[0] & _A[1])?D0:0;
  assign #4 w1 = (E &  A[0] & _A[1])?D1:0;
  assign #4 w2 = (E & _A[0] &  A[1])?D2:0;
  assign #4 w3 = (E &  A[0] &  A[1])?D3:0;
  assign #4 Y  = w0 | w1 | w2 | w3;
endmodule
```

3.5 Binary encoder

A ***binary encoder*** is a combinational circuit having ***n*** output lines and 2^n input lines, where only an input line at a time can be asserted and the value obtained from the output reports the number of the asserted input line. The following examples show binary encoders with ***n*** equal to 1, 2 and 3.

Table 3.29. Truth table for a binary encoder with two inputs and one output.

D_1	D_0	Y_0
0	1	0
1	0	1

Figure 3.30. To obtain a binary encoder with two input lines it is enough to connect the second input line to the output.

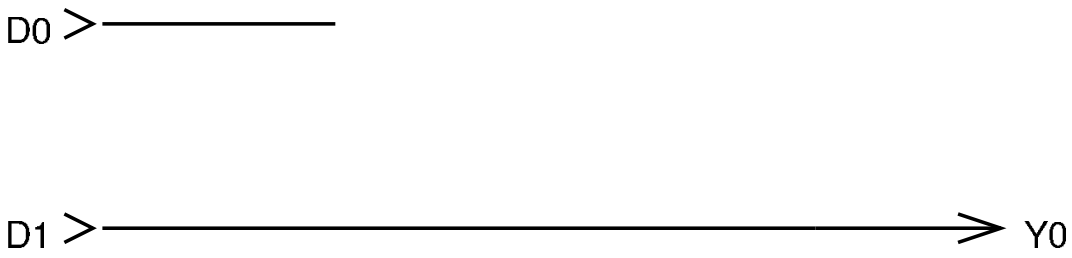


Table 3.31. Truth table for a binary encoder with four input and two output ports.

D_3	D_2	D_1	D_0	Y_1	Y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Figure 3.32. Binary encoder with four input and two output ports. The first input line is not connected, because if it is asserted the output should be zero anyway.

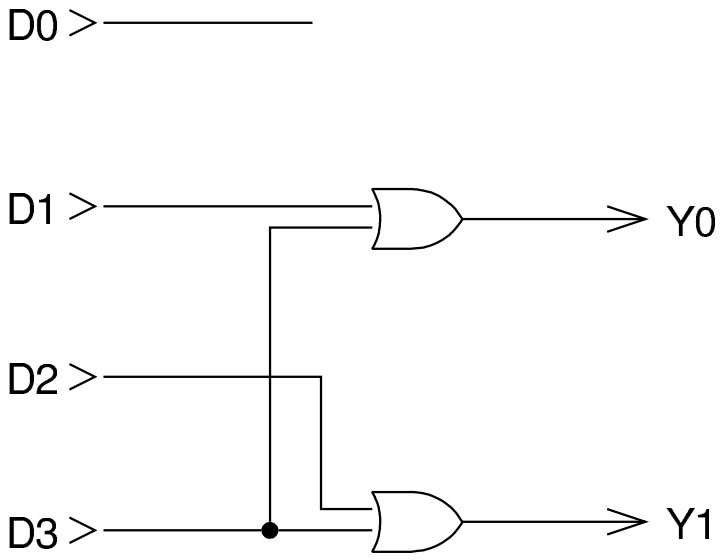
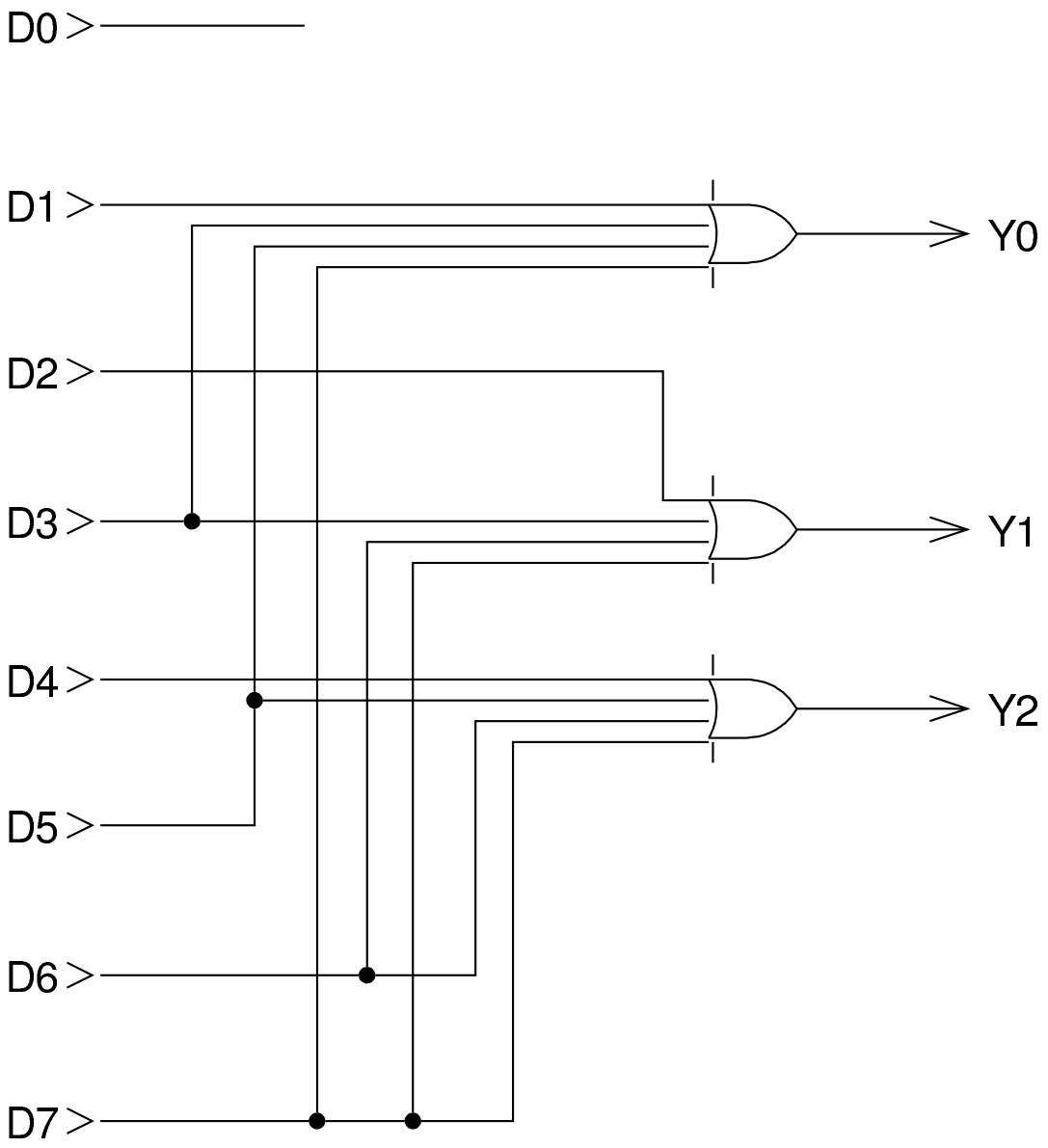


Table 3.33. Truth table for a binary encoder with eight input and three output ports.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Figure 3.34. Binary encoder with eight input and three output ports.



Listing 3.35. Binary encoder example, with eight input and three output ports, written in Verilog. This solution is different from the picture 3.34, because if the input value is not as expected, the encoded output is zero.

```
module binary_encoder (Y, D);
  input [7:0] D;
  output [2:0] Y;
  //
  function [2:0] be8to3 (input [7:0] D);
    case (D)
        1: be8to3 = 0;
```

```
          2: be8to3 = 1;
          4: be8to3 = 2;
          8: be8to3 = 3;
         16: be8to3 = 4;
         32: be8to3 = 5;
         64: be8to3 = 6;
        128: be8to3 = 7;
        default: be8to3 = 0;
      endcase
    endfunction
    //
    assign #6 Y = be8to3 (D);
endmodule
```

Listing 3.36. Binary encoder example describing the gates in picture 3.34.

```
module binary_encoder (Y, D);
  input [7:0] D;
  output [2:0] Y;
  assign #6 Y[0] = D[1] | D[3] | D[5] | D[7];
  assign #6 Y[1] = D[2] | D[3] | D[6] | D[7];
  assign #6 Y[2] = D[4] | D[5] | D[6] | D[7];
endmodule
```

3.6 Priority encoder

«

The ***priority encoder*** is a binary encoder that allows the assertion of every input line, giving on the output the number corresponding to the asserted input having priority above all the others. In that case it is also allowed to have no input line asserted, thus there might be an additional output port that is asserted only when there is at least an input line asserted.

Table 3.37. Truth table for a priority encoder with four input ports. The asserted input with the lower index has priority above the other, therefore the letter 'X' is used to show an unspecified value for the other lower priority input ports. The output port ***GS*** (*group select*) is asserted when at least an input line is asserted.

D_3	D_2	D_1	D_0	GS	Y_1	Y_0
0	0	0	0	0	X	X
X	X	X	1	1	0	0
X	X	1	0	1	0	1
X	1	0	0	1	1	0
1	0	0	0	1	1	1

Figure 3.38. Priority encoder with four input ports.

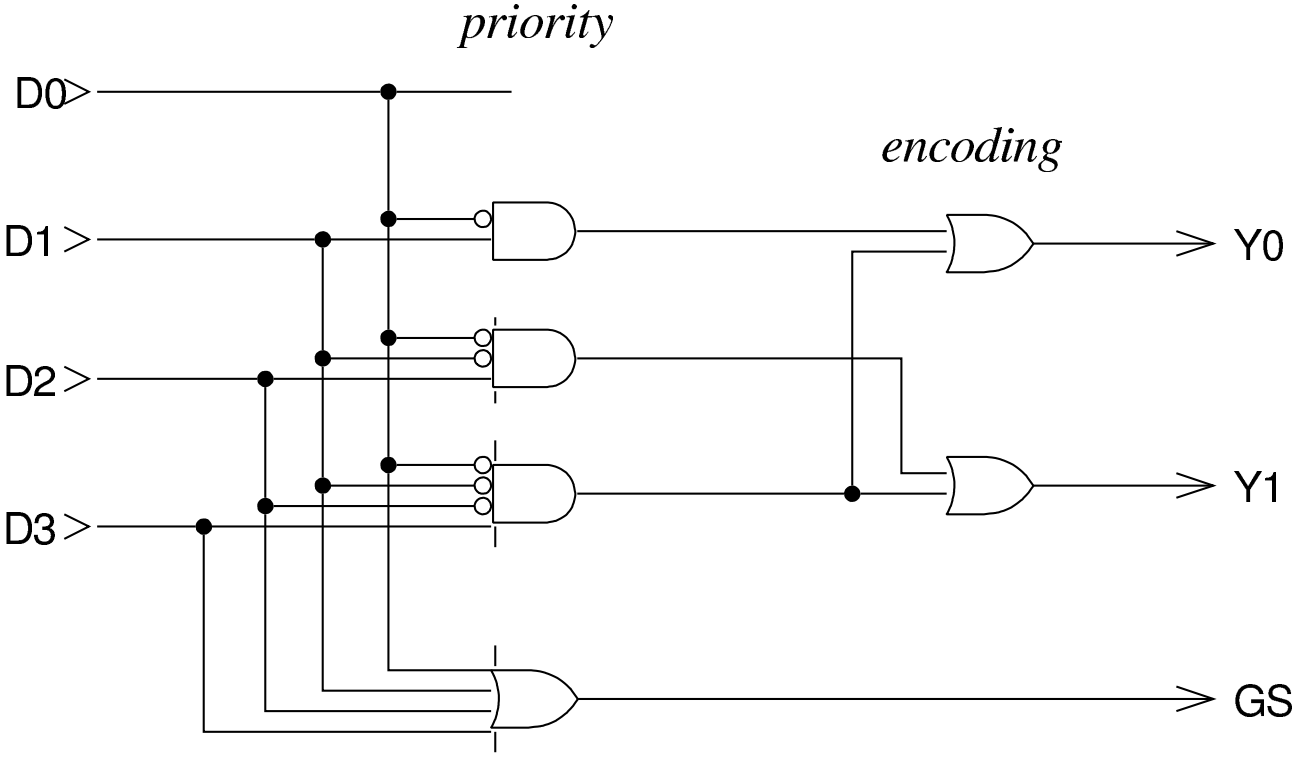
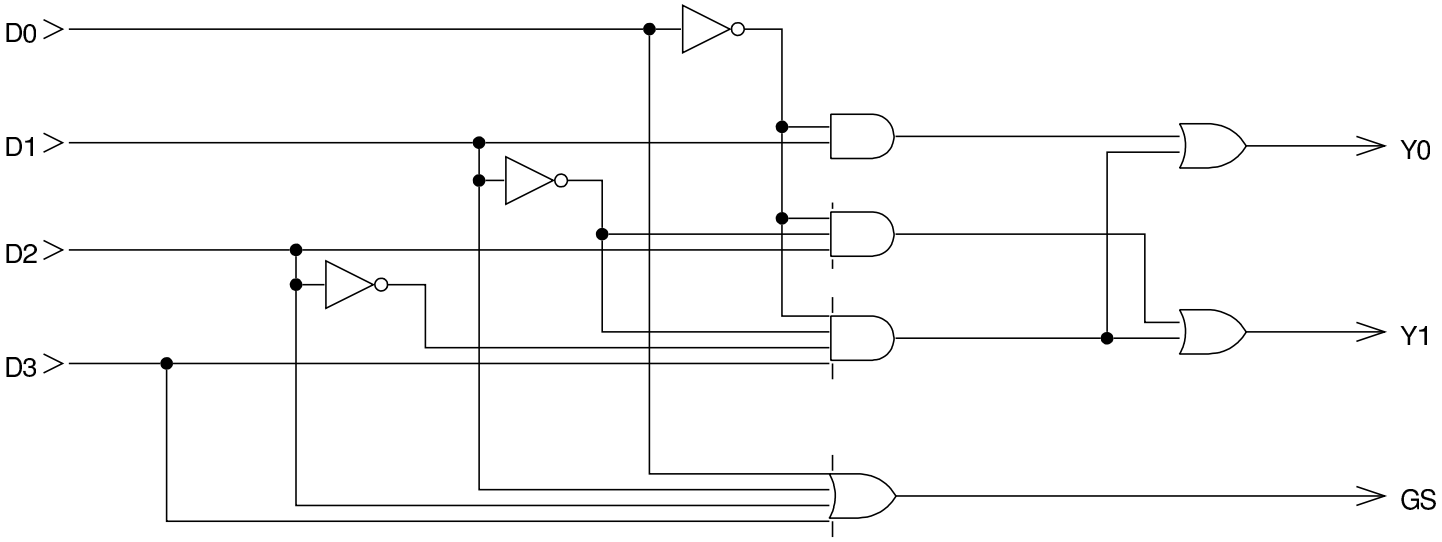


Figure 3.39. A different way to obtain the same priority encoder with four input ports.



Listing 3.40. Example with Verilog.

```
module priority_encoder_4_to_2 (Y, GS, D);
  input [3:0] D;
  output [1:0] Y;
  output GS;
  //
  function [1:0] f4to2 (input [3:0] D);
    if (D & 1)
      f4to2 = 0;
    else if (D & 2)
      f4to2 = 1;
    else if (D & 4)
      f4to2 = 2;
    else if (D & 8)
      f4to2 = 3;
    else
      f4to2 = 0;
  endfunction
  //
  assign #8  Y = f4to2 (D);
  assign #8 GS = D[0] | D[1] | D[2] | D[3];
```

```
endmodule
```

Listing 3.41. Verilog gate level alternative code following the picture 3.38.

```
module priority_encoder_4_to_2 (Y, GS, D);
  input [3:0] D;
  output [1:0] Y;
  output GS;
  wire w0, w1, w2;
  assign #4 w0 = ~D[0] & D[1];
  assign #4 w1 = ~D[0] & ~D[1] & D[2];
  assign #4 w2 = ~D[0] & ~D[1] & ~D[2] & D[3];
  assign #4 Y[0]  = w0 | w2;
  assign #4 Y[1]  = w1 | w2;
  assign #8 GS = D[0] | D[1] | D[2] | D[3];
endmodule
```

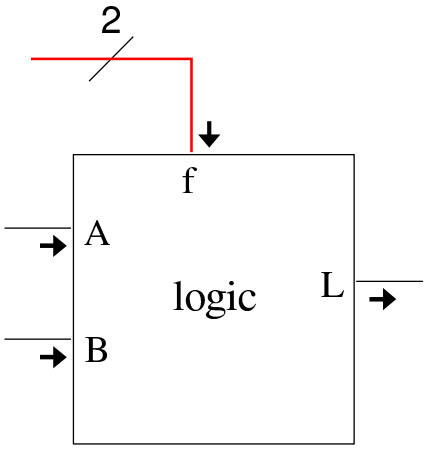
Listing 3.42. Verilog gate level alternative code following the picture 3.39.

```
module priority_encoder_4_to_2 (Y, GS, D);
  input [3:0] D;
  output [1:0] Y;
  output GS;
  wire [2:0] _D;
  wire w0, w1, w2;
  assign _D = ~D;
  assign #4 w0 = _D[0] & D[1];
  assign #4 w1 = _D[0] & _D[1] & D[2];
  assign #4 w2 = _D[0] & _D[1] & _D[2] & D[3];
  assign #4 Y[0]  = w0 | w2;
  assign #4 Y[1]  = w1 | w2;
  assign #8 GS = D[0] | D[1] | D[2] | D[3];
endmodule
```

A common priority encoder circuit is 74148 (see i1.3.1).

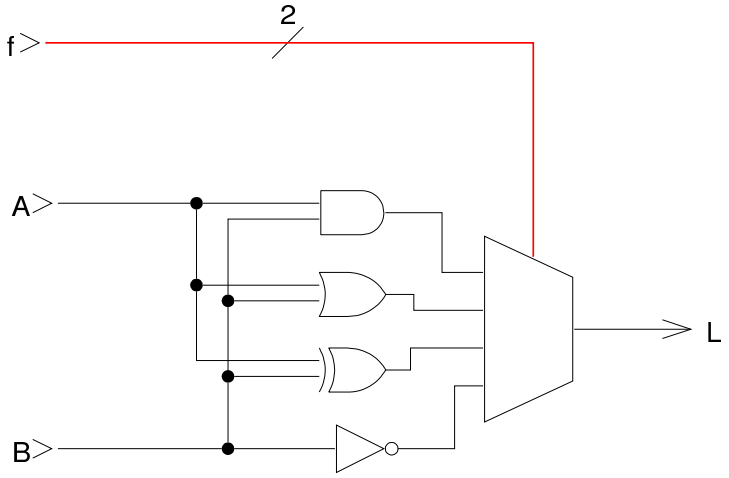
3.7 Logic units

Using a multiplexer it is possible to obtain easily a logic unit, able to do the common logic operations, selecting the desired one through a selection value. For example, it might be possible to build the circuit that does the task shown in the following figure:



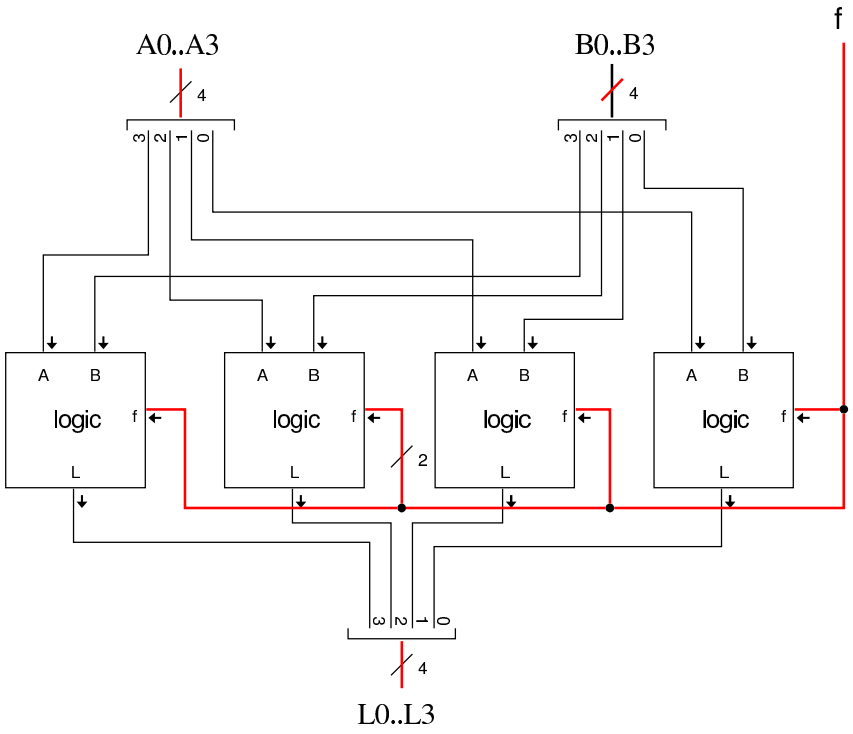
f_1	f_0	output
0	0	A AND B
0	1	A OR B
1	0	A XOR B
1	1	NOT B

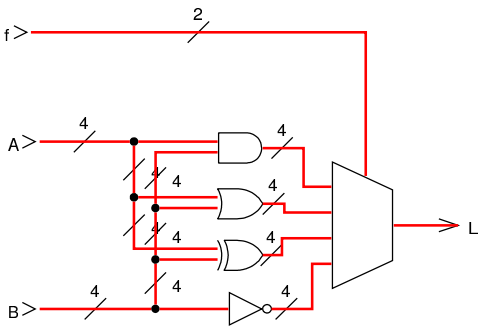
Figure 3.44. Logic unit implementation example with four options.



Logic units like this one are useful if the input lines (***A*** and ***B***) are multi-bit sized. To obtain this it suffice to place in parallel more single-bit units with the function control input (***f***) connected together.

Figure 3.45. More logic units can be connected in paralel, to operate on multi-bit values. The diagram on the right is a compact representation of the same circuit.





f_1	f_0	output
0	0	A AND B
0	1	A OR B
1	0	A XOR B
1	1	NOT B

3.8 Shift and rotation

The bit shift is an important binary operation that can be implemented with a combinational circuit. There is an ordered group of ***n*** inputs, starting from 0 to ***n***−1, a group of ***n*** outputs ordered in the same way and there might be a carry input and a carry output. Often it is considered the shift of a single binary digit, because more binary digits might be shifted repeating the operation. The shift and rotation operations require to specify the direction: left or right.

The easiest kind of shift is the ***logic shift***, which moves the bits left or right, losing the most significant bit or the least significant one, inserting a zero at the opposite side. The following figures implements the logic shift and the other types of shifts and rotations, using multiplexers, where the connection to the ground means a fixed 0 input.

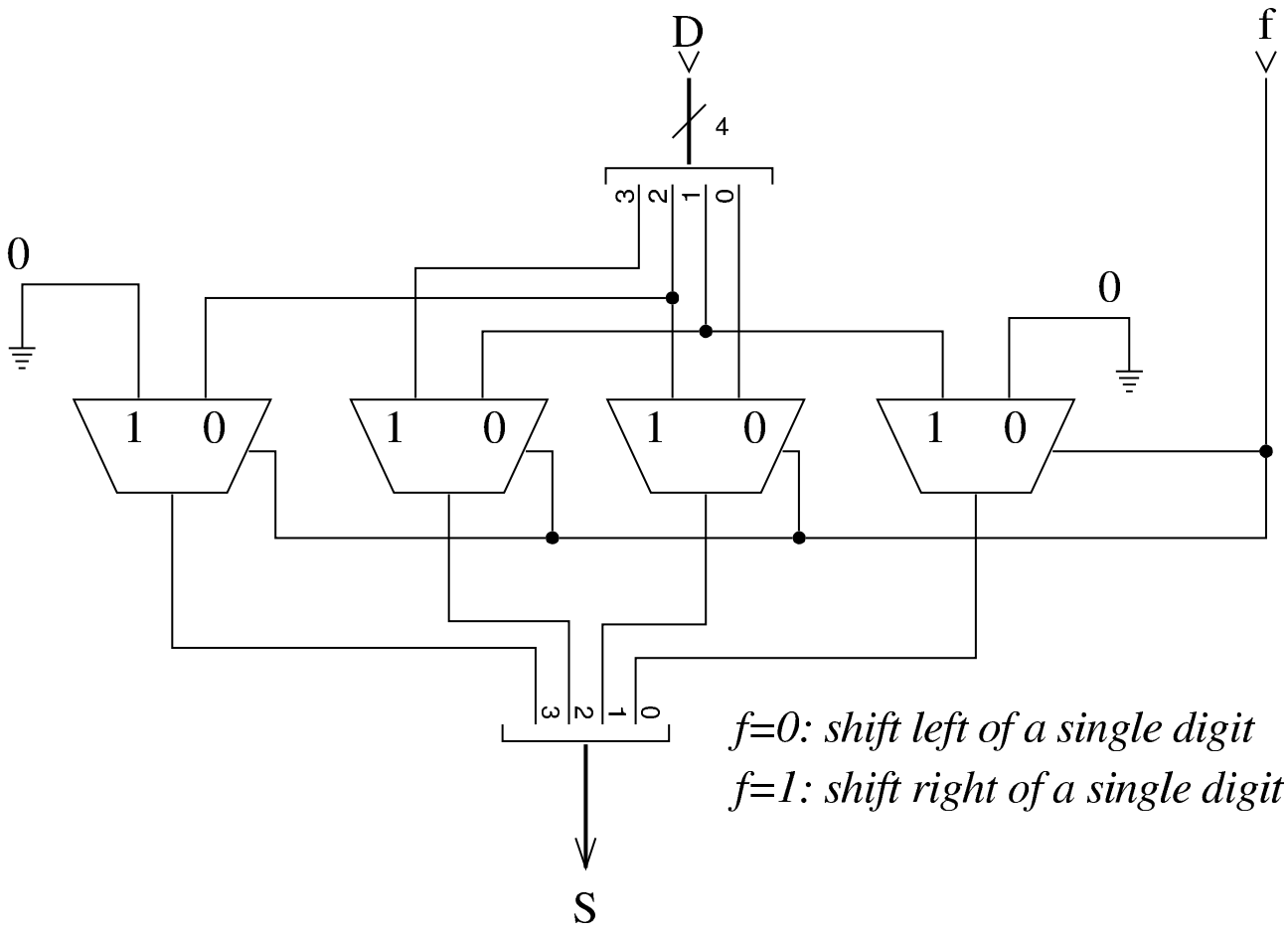
1 1 1 0 *original value*

1 1 0 0 *left logic shift*

1 1 1 0 *original value*

0 1 1 1 *right logic shift*

Figure 3.47. 4-digit left and right logic shift, with 2-input multiplexers. The shift direction is established by the value received by the ***f*** input. The shift loses a binary digit from one side and gets a zero from the opposite side.

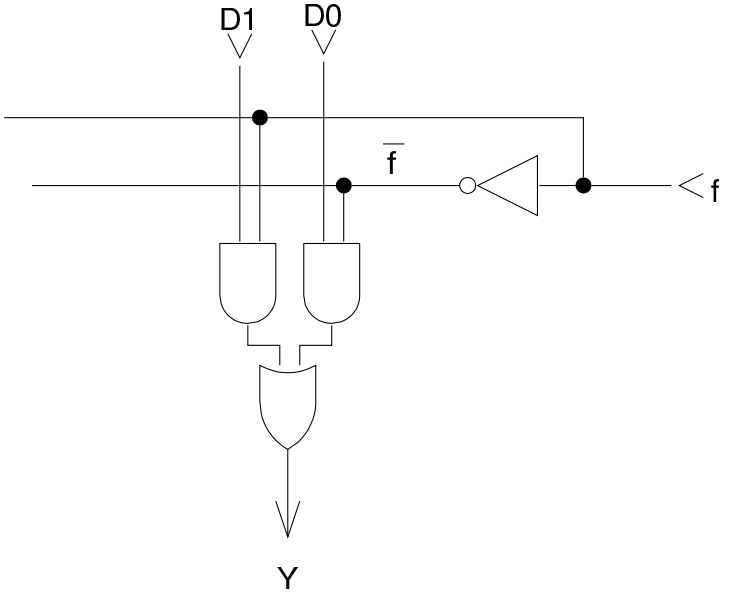


Listing 3.48. 4-digit logic shift: example with Verilog.

```
module logic_shift_4 (S, D, f);
  input [3:0] D;
  input f;
  output [3:0] S;
  //
  function [3:0] lsh (input [3:0] D, input f);
    if (f == 0)
      lsh = D << 1;
    else
      lsh = D >> 1;
  endfunction
  //
```

```
  assign #6 S = lsh (D, f);
endmodule
```

The other Verilog examples of the section describe only the gate logic level, assuming that the 2-input multiplexers are implemented as the following figure shows:



Listing 3.50. 4-digit logic shift: detailed gate level implementation.

```
module logic_shift_4 (S, D, f);
  input [3:0] D;
  input f;
  output [3:0] S;
  wire _f = ~f;
  //
  assign #6 S[0] = (D[1] & f);
  assign #6 S[1] = ((D[2] & f) | (D[0] & _f));
  assign #6 S[2] = ((D[3] & f) | (D[1] & _f));
  assign #6 S[3] = (D[2] & _f);
endmodule
```

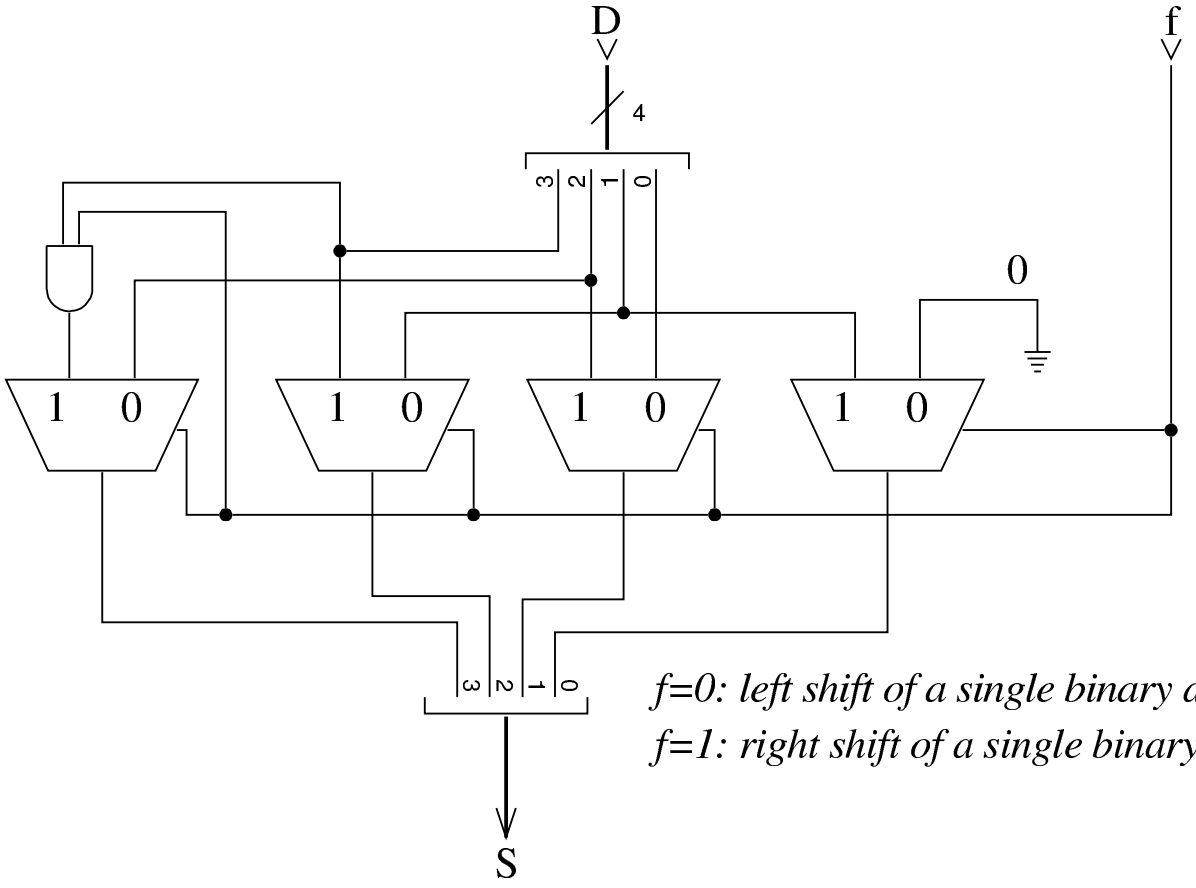
The ***arithmetic shift*** is made in the same way as the logic one, with the exception that the right shift must keep the sign unaltered. In practice, when doing an arithmetic right shift, the digit inserted on the most significant position is the same as the original one. This kind of shift is called ‘arithmetic’ because the result is the same as the

multiplication and division by two. However it should be noticed that the left shift might produce an overflow, corresponding to the lost of the original sign.

1	1	1	0	*original value*	1	1	1	0	*original value*
1	1	0	0	*left arithmetic shift*	1	1	1	1	*right arithmetic shift*
0	1	1	0	*original value*	0	1	1	0	*original value*
1	1	0	0	*left arithmetic shift (*)*	0	0	1	1	*right arithmetic shift*
1	0	1	0	*original value*	0	1	1	0	*original value*
0	1	0	0	*left arithmetic shift (*)*	0	0	1	1	*right arithmetic shift*

(*) when shifting to the left, a sign inversion might be produced (overflow)

Figure 3.52. 4-digit, left and right arithmetic shift, using 2-input multiplexers. The shift direction is defined by the value received by the ***f*** input.



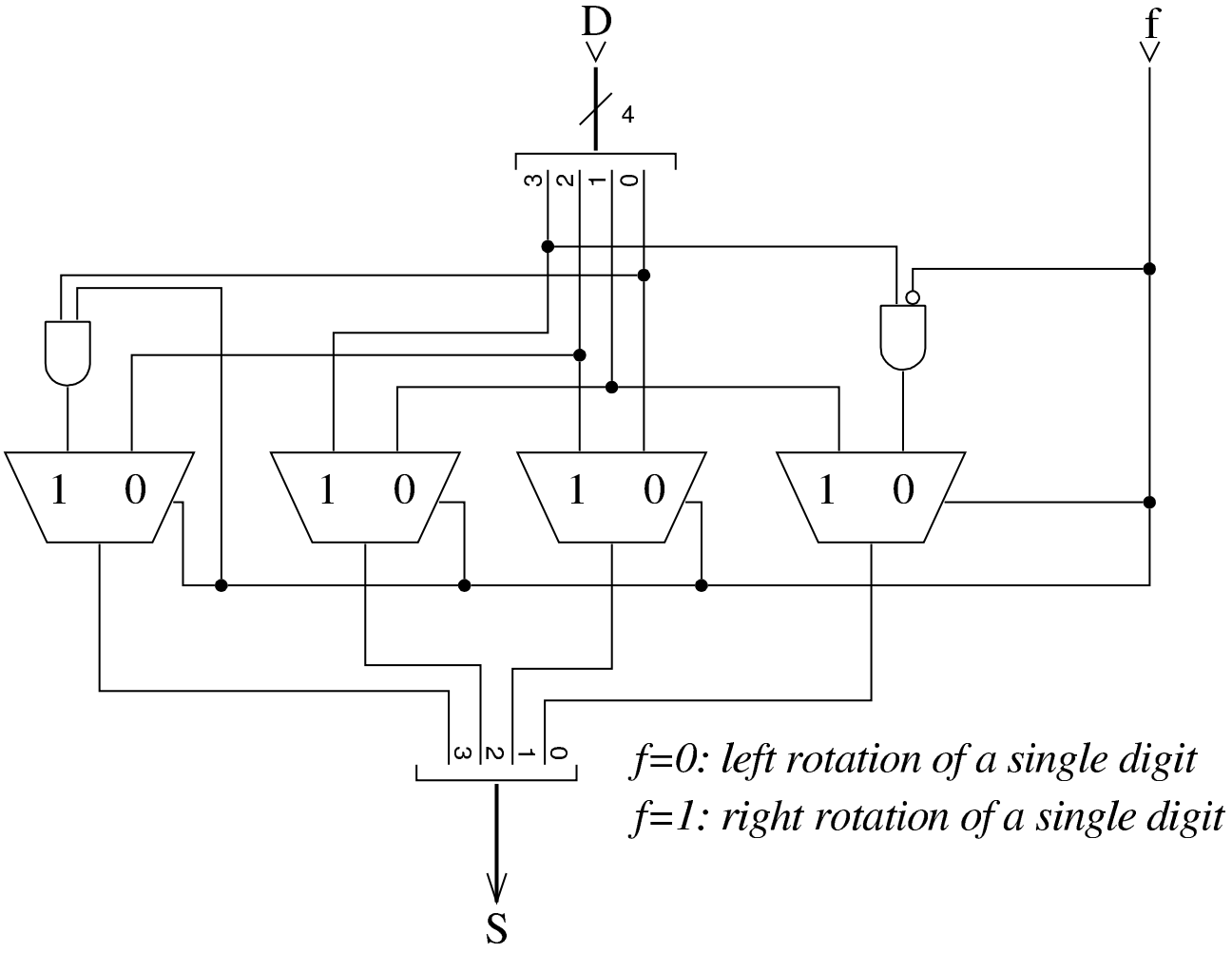
Listing 3.53. 4-digit arithmetic shift: detailed gate level implementation.

```
module arithmetic_shift_4 (S, D, f);
  input [3:0] D;
  input f;
  output [3:0] S;
  wire _f = ~f;
  //
  assign #6 S[0] = (D[1] & f);
  assign #6 S[1] = ((D[2] & f) | (D[0] & _f));
  assign #6 S[2] = ((D[3] & f) | (D[1] & _f));
  assign #6 S[3] = ((D[3] & f) | (D[2] & _f));
endmodule
```

The ***rotation*** is a shift operation where the binary digit lost from one side is inserted on the opposite side, either on the left or the right rotation.

1	1	1	0	*original value*		1	1	1	0	*original value*
1	1	0	1	*left rotation*		0	1	1	1	*right rotation*
0	1	1	0	*original value*		0	1	1	0	*original value*
1	1	0	0	*left rotation*		0	0	1	1	*right rotation*
1	0	1	0	*original value*		1	0	1	0	*original value*
0	1	0	1	*left rotation*		0	1	0	1	*right rotation*

Figure 3.55. 4-digit rotation, with 2-input multiplexers. The rotation direction is specified by the value received by the input ***f***.



Listing 3.56. 4-digit rotation: detailed gate level implementation.

```
module rotation_4 (S, D, f);
  input [3:0] D;
  input f;
  output [3:0] S;
  wire _f = ~f;
  //
  assign #6 S[0] = ((D[1] & f) | (D[3] & _f));
  assign #6 S[1] = ((D[2] & f) | (D[0] & _f));
  assign #6 S[2] = ((D[3] & f) | (D[1] & _f));
  assign #6 S[3] = ((D[0] & f) | (D[2] & _f));
endmodule
```

The ***rotation with carry*** uses the previous carry for the digit inserted on one side and saves the removed digit from the opposite side as the new value for the carry (after the rotation).

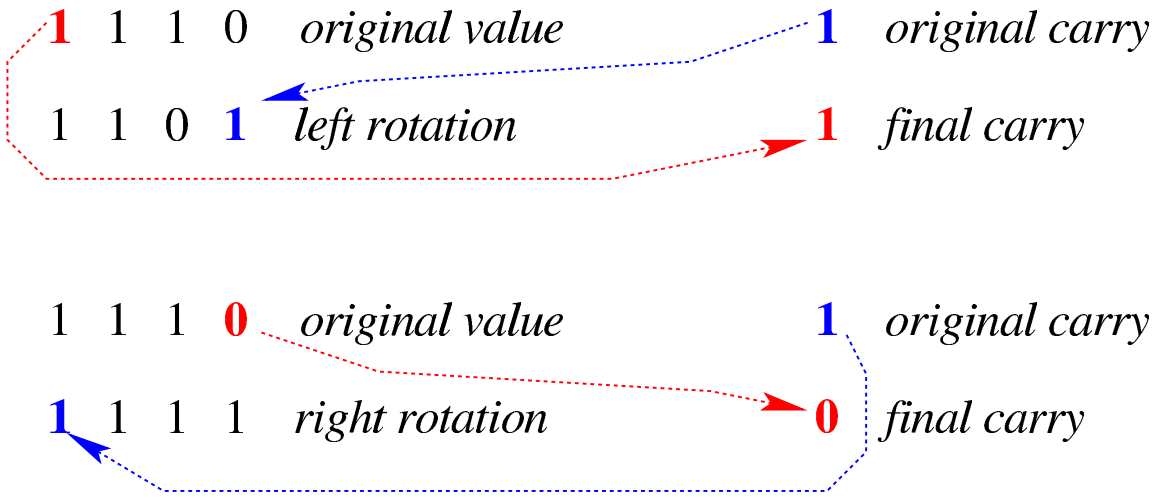
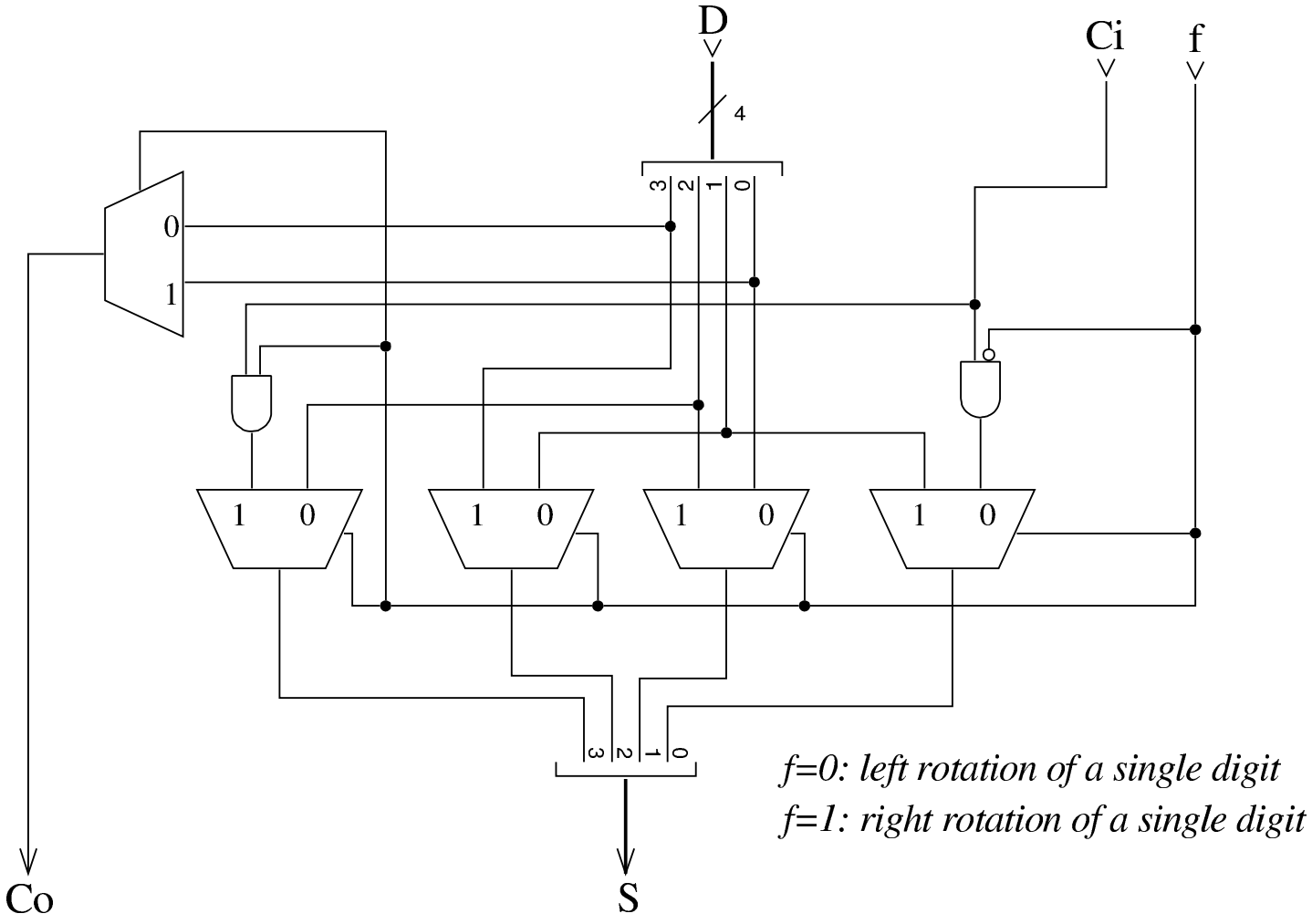


Figure 3.58. 4-digit input, rotation with carry, using 2-input multiplexers. The rotation direction is specified by the value received by the ***f*** input.



3.9 Addition

«

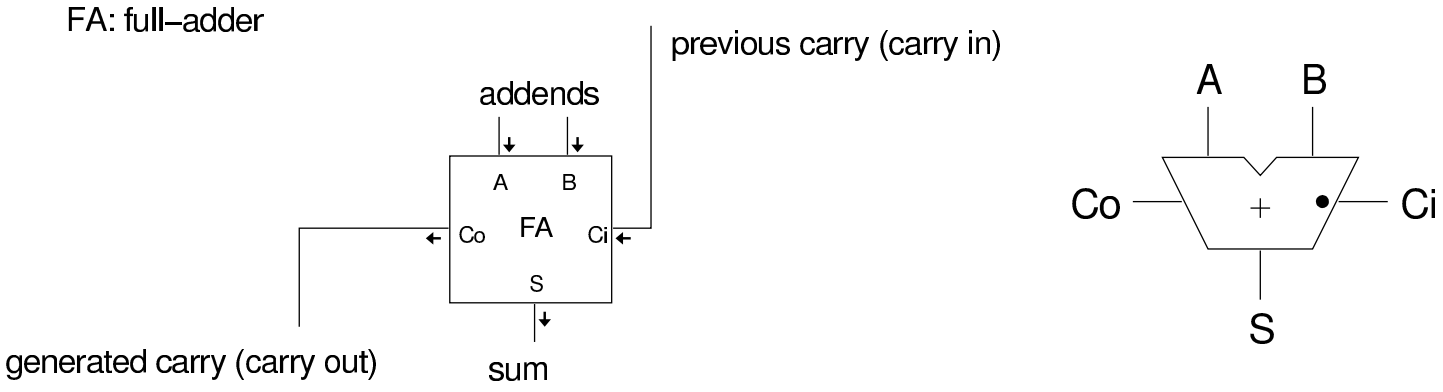
The binary addition, made in the same common way used for the decimal numbering system, does not generate a carry higher than 1. It can be verified through the following table.

Table 3.59. Binary addition.

first addend	second addend	previous carry (carry in)	generated carry (carry out)	result
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Therefore, the combinational circuit that can do the binary addition must have three input (first addend, second addend, carry in) and two output ports (result, carry out).

Figure 3.60. Adder block diagram. The abbreviation FA stands for ***full-adder***.



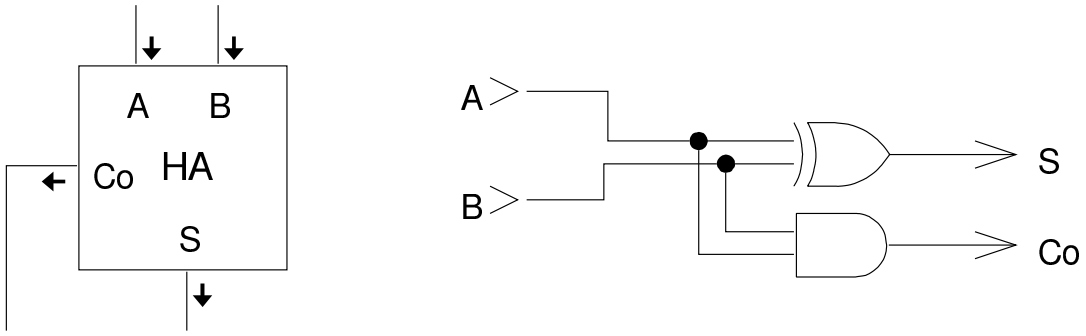
It is custom to consider the ***full-adder*** (the one shown above) to be made of two smaller modules, known as ***half-adders***. The half-adder is a combinational circuit with two input ports, corresponding to the two addends, and two output ports, corresponding to the sum and the generated carry; in practice, the half-adder has no previous carry.

Table 3.61. Truth table for a half-adder: ***A*** and ***B*** are the addends, ***S*** and ***Co*** are the sum and the output carry.

A	***B***	***Co***	***S***
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

It can be easily guessed that the half-adder sum can be obtained with an XOR gate and that the output carry requires an AND gate.

Figure 3.62. Half-adder block diagram and implementation example. The abbreviation HA stands for ***half-adder***.



To obtain a full-adder from half-adders, it is necessary to sum also the input carry.

Figure 3.63. Full-adder block diagram, obtained with two half-adders.

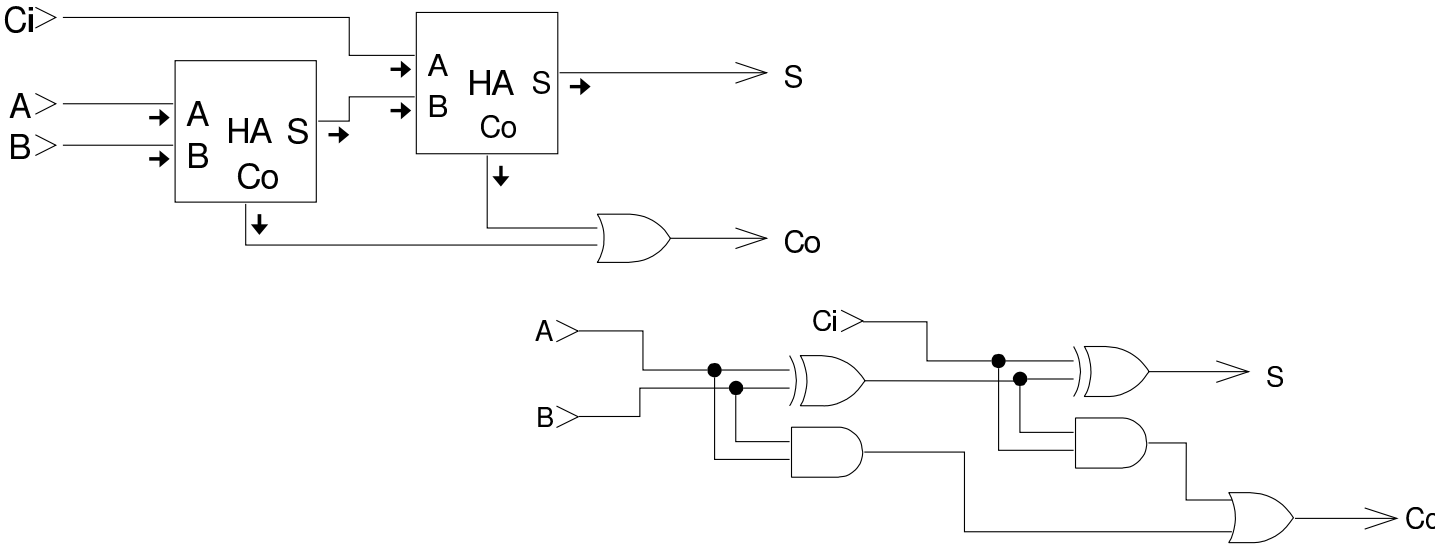
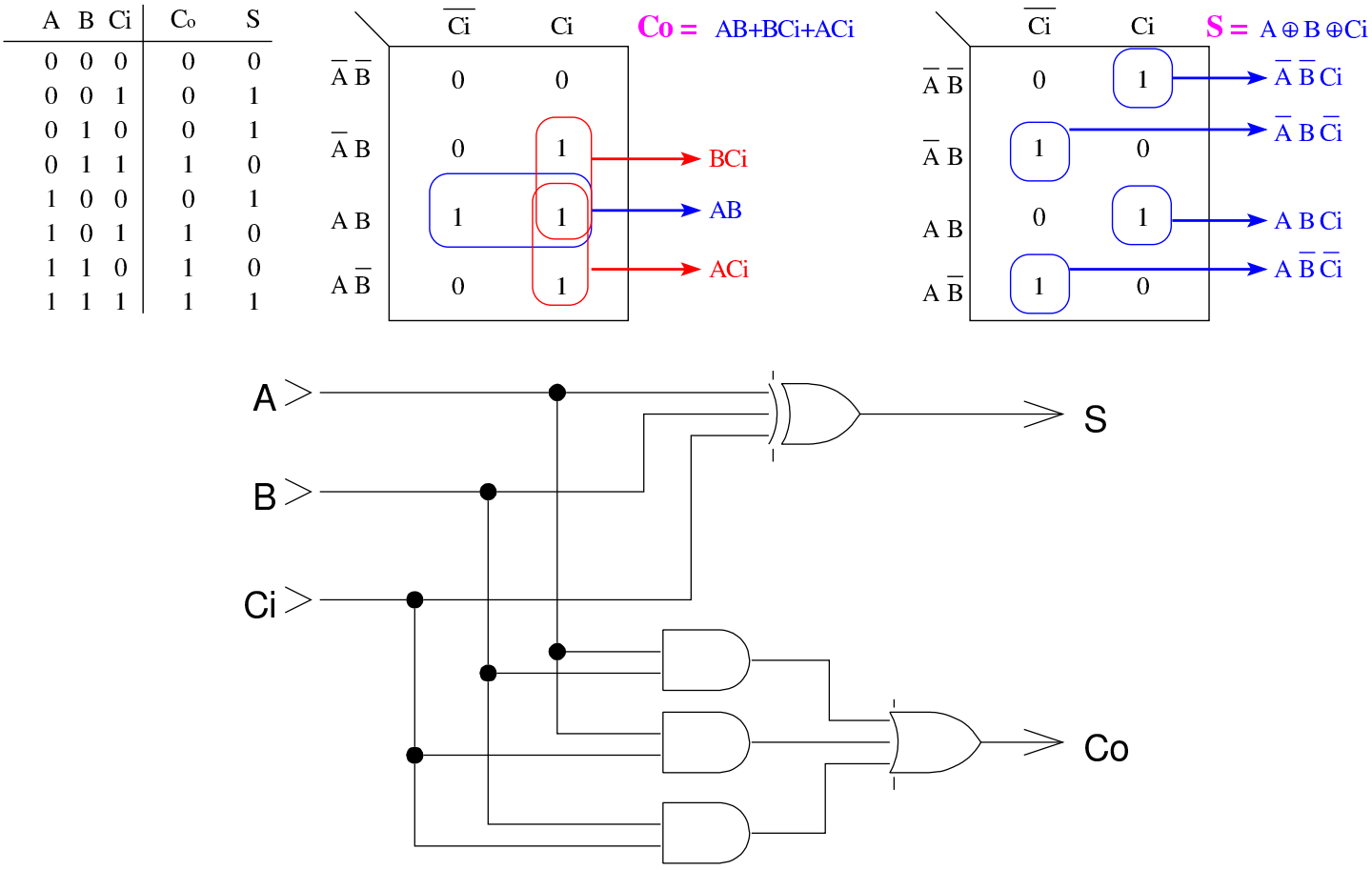
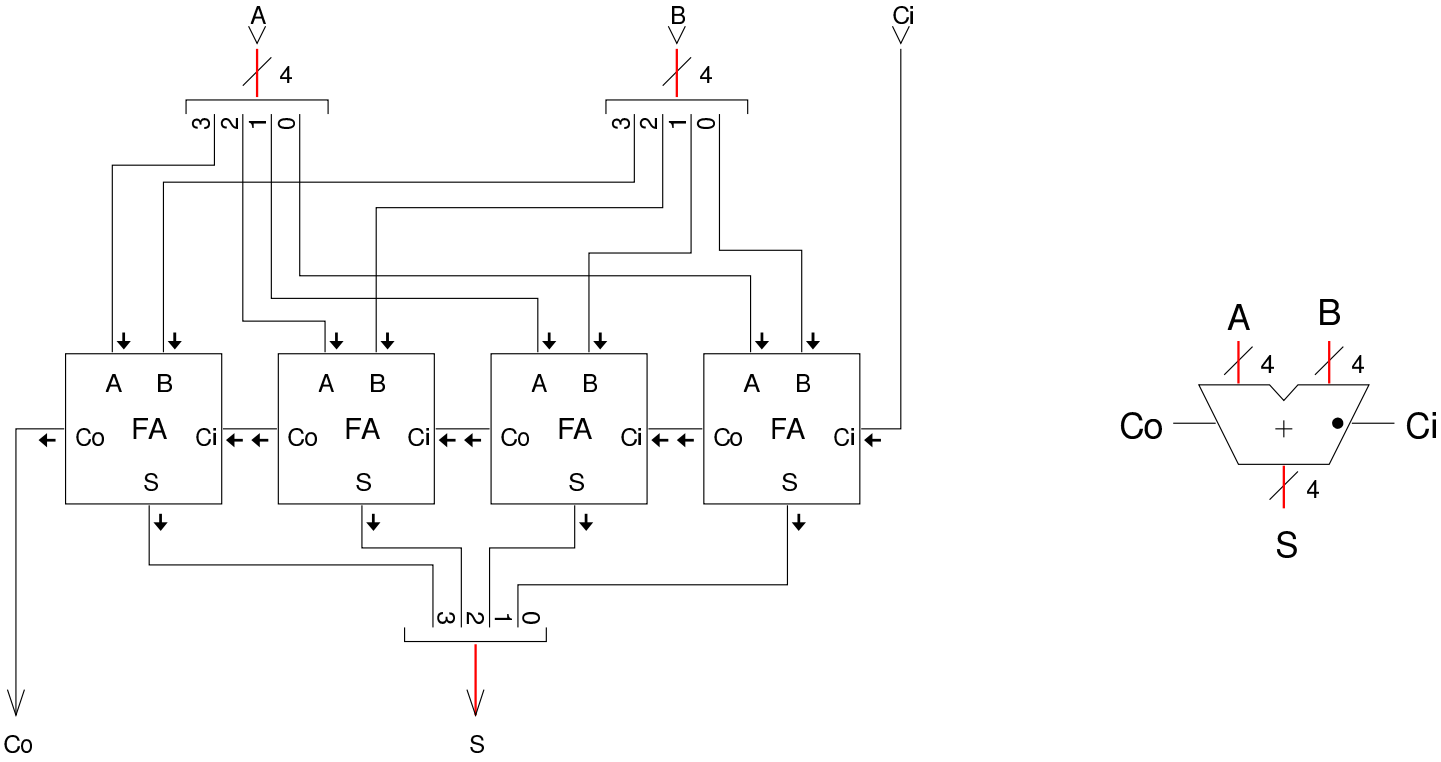


Figure 3.64. Full-adder alternative implementations, with the help of Karnaugh maps.



To be useful, the addition should be applied to binary addends composed of more than a single digit, therefore it is necessary to assemble in paralel some adders, passing properly the carry from the less significant digit to the more significant one, digit by digit. This method of carry propagation is known as ***ripple-carry*** and is illustrated by the following figure.

Figure 3.65. Ripple-carry 4-digit adder. On the right is shown a compact representation, but it is not specified there in what way the carry is propagated.



The described adder works correctly with the addition of positive and negative numbers, when the negative numbers are represented as 2's complement of the positive equivalent value.

Listing 3.66. A simple Verilog implementation of a 4-bit adder.

```
module adder_4 (Co, S, A, B, Ci);
  input [3:0] A, B;
  input Ci;
  output Co;
  output [3:0] S;
  //
  assign #30 {Co, S} = A + B + Ci;
endmodule
```

Listing 3.67. Gate level implementation of a 4-bit adder with ripple-carry.

```
module adder_4 (Co, S, A, B, Ci);
  input [3:0] A, B;
  input Ci;
  output Co;
  output [3:0] S;
  wire [4:0] C;
  //
  function add (input A, input B, input Ci);
    add = A^B^Ci;
  endfunction
  //
  function carry (input A, input B, input Ci);
    carry = A&B|A&Ci|B&Ci;
  endfunction
  //
  assign #0 C[0] = Ci;
  assign #6 C[1] = carry (A[0], B[0], C[0]);
  assign #6 C[2] = carry (A[1], B[1], C[1]);
  assign #6 C[3] = carry (A[2], B[2], C[2]);
  assign #6 C[4] = carry (A[3], B[3], C[3]);
  //
  assign #5 S[0] = add (A[0], B[0], C[0]);
  assign #5 S[1] = add (A[1], B[1], C[1]);
  assign #5 S[2] = add (A[2], B[2], C[2]);
  assign #5 S[3] = add (A[3], B[3], C[3]);
  //
  assign #0 Co = C[4];
endmodule
```

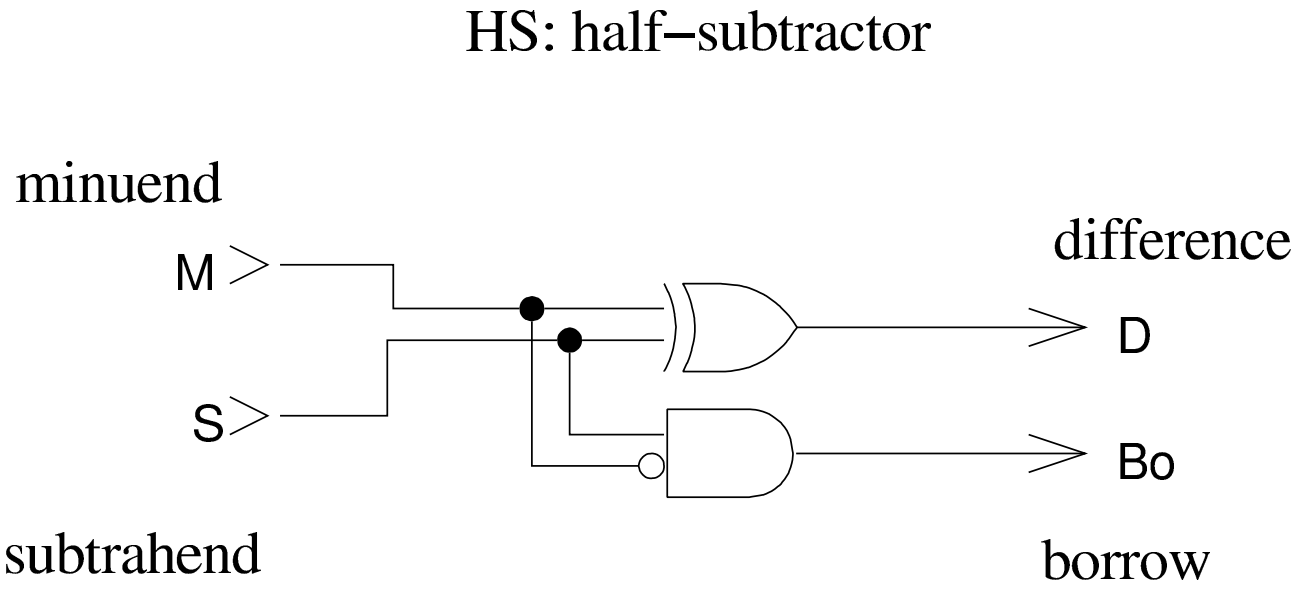
3.10 Subtraction

The basic building block for the subtraction is the ***half-subtractor***. As the order of the operands is significant, it is necessary to distinguish between minuend and subtrahend, therefore all the following examples will use this notation: $D=M-S$.

Table 3.68. Half-subtractor truth table.

M	S	B_o	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

Figure 3.69. Half-subtractor.



Instead of the output carry there is the ***borrow*** (output borrow) that is asserted when the subtrahend is greater than the minuend. The ***full-subtractor*** has another input, which is the input borrow coming from a previous digit. To build the full-subtractor it is possible to use two half-subtractors, where the second one subtracts from the result of the first one the input borrow.

Table 3.70. Full-subtractor truth table.

M	S	B_i	B_o	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Figure 3.71. Full-subtractor.

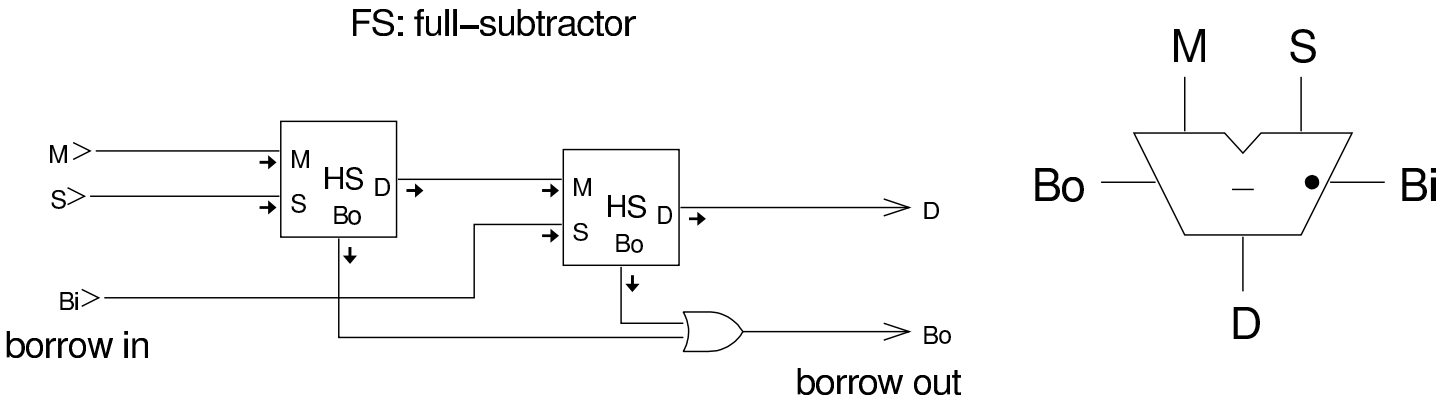
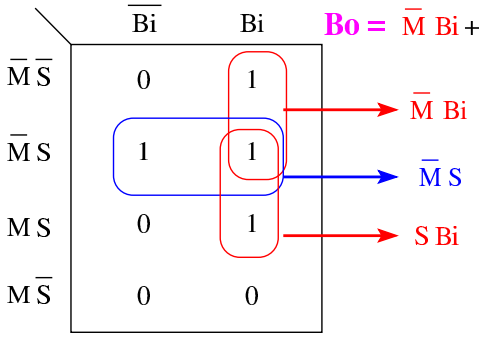
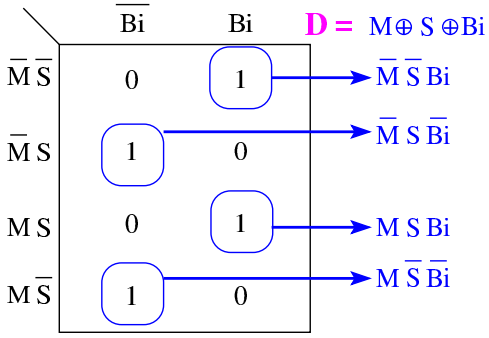
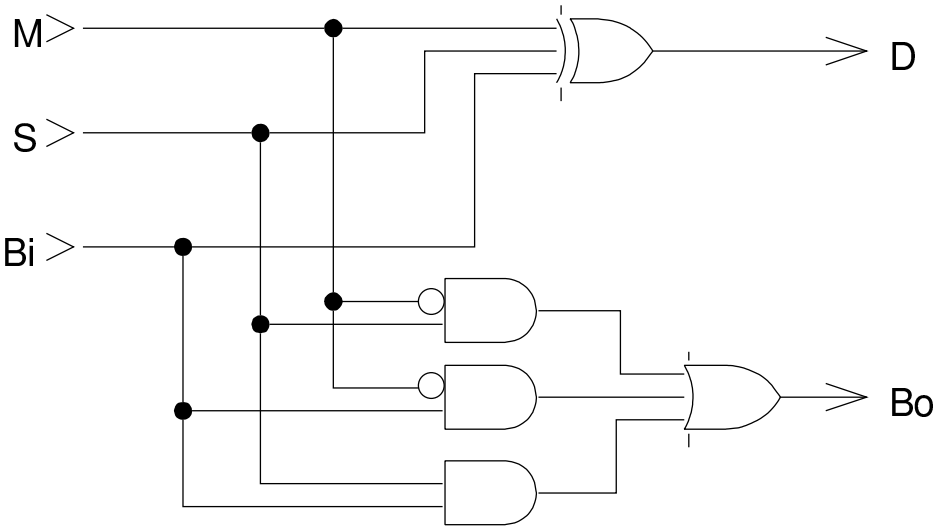


Figure 3.72. Full-subtractor alternative implementation, with the help of Karnaugh maps.

M	S	Bi	B_o	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

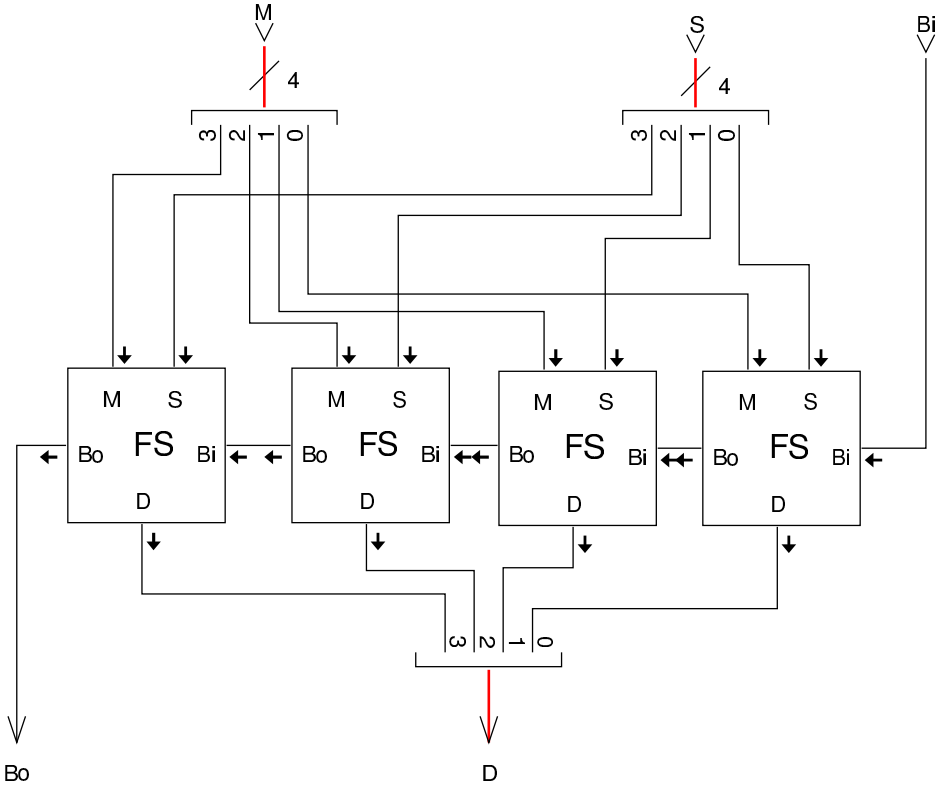


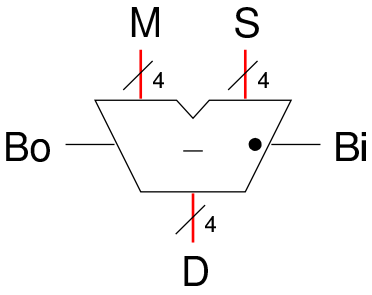




As for the addition, it is possible to build a chain of full-subtractor.

Figure 3.73. 4-bit subtraction.





Listing 3.74. A simple Verilog implementation of a 4-bit subtractor.

```
module subtractor_4 (Bo, D, M, S, Bi);
  input [3:0] M, S;
  input Bi;
  output Bo;
  output [3:0] D;
  //
  assign #30 {Bo, D} = M - S - Bi;
endmodule
```

Listing 3.75. Gate level implementation of a 4-bit subtractor.

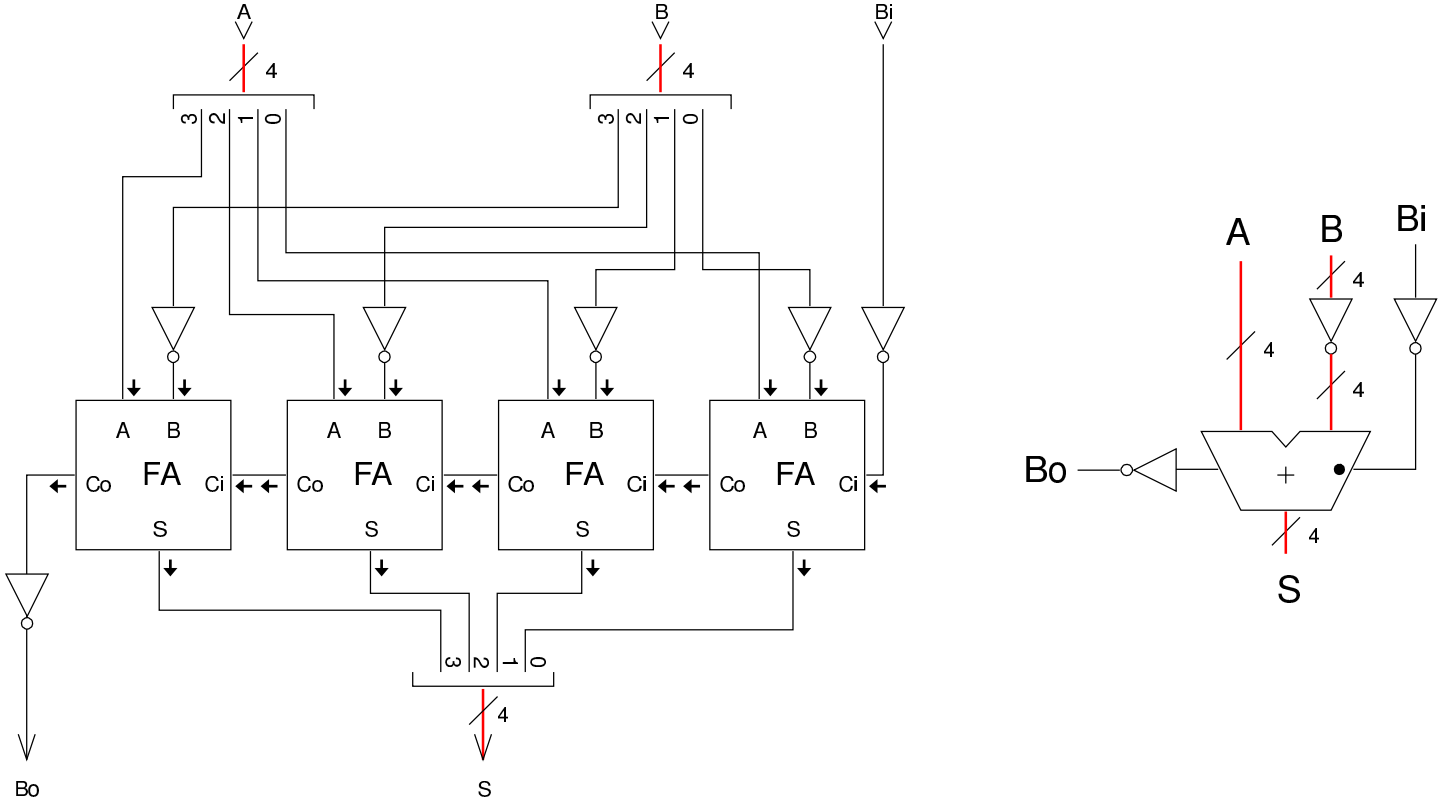
```
module subtractor_4 (Bo, D, M, S, Bi);
  input [3:0] M, S;
  input Bi;
  output Bo;
  output [3:0] D;
  wire [4:0] B;
  //
  function sub (input M, input S, input Bi);
    sub = M^S^Bi;
  endfunction
  //
  function borrow (input M, input S, input Bi);
    borrow = ~M&Bi|S&Bi|~M&S;
  endfunction
  //
  assign #0 B[0] = Bi;
  assign #6 B[1] = borrow (M[0], S[0], B[0]);
  assign #6 B[2] = borrow (M[1], S[1], B[1]);
  assign #6 B[3] = borrow (M[2], S[2], B[2]);
  assign #6 B[4] = borrow (M[3], S[3], B[3]);
  //
  assign #5 D[0] = sub (M[0], S[0], B[0]);
  assign #5 D[1] = sub (M[1], S[1], B[1]);
```

```
   assign #5 D[2] = sub (M[2], S[2], B[2]);
   assign #5 D[3] = sub (M[3], S[3], B[3]);
   //
   assign #0 Bo = B[4];
endmodule
```

3.11 Addition and subtraction together

The full-adder works correctly for the sum of positive and negative numbers, when the negative ones are represented by the 2's complement method, therefore, to transform an adder into a subtractor, it is enough to calculate the 2's complement for the subtrahend. To achieve the result it is more convenient to invert all the subtrahend bits (1's complement) and also the input carry, which now assumes the role of input borrow. To obtain coherently an output borrow, even the output carry should be inverted.

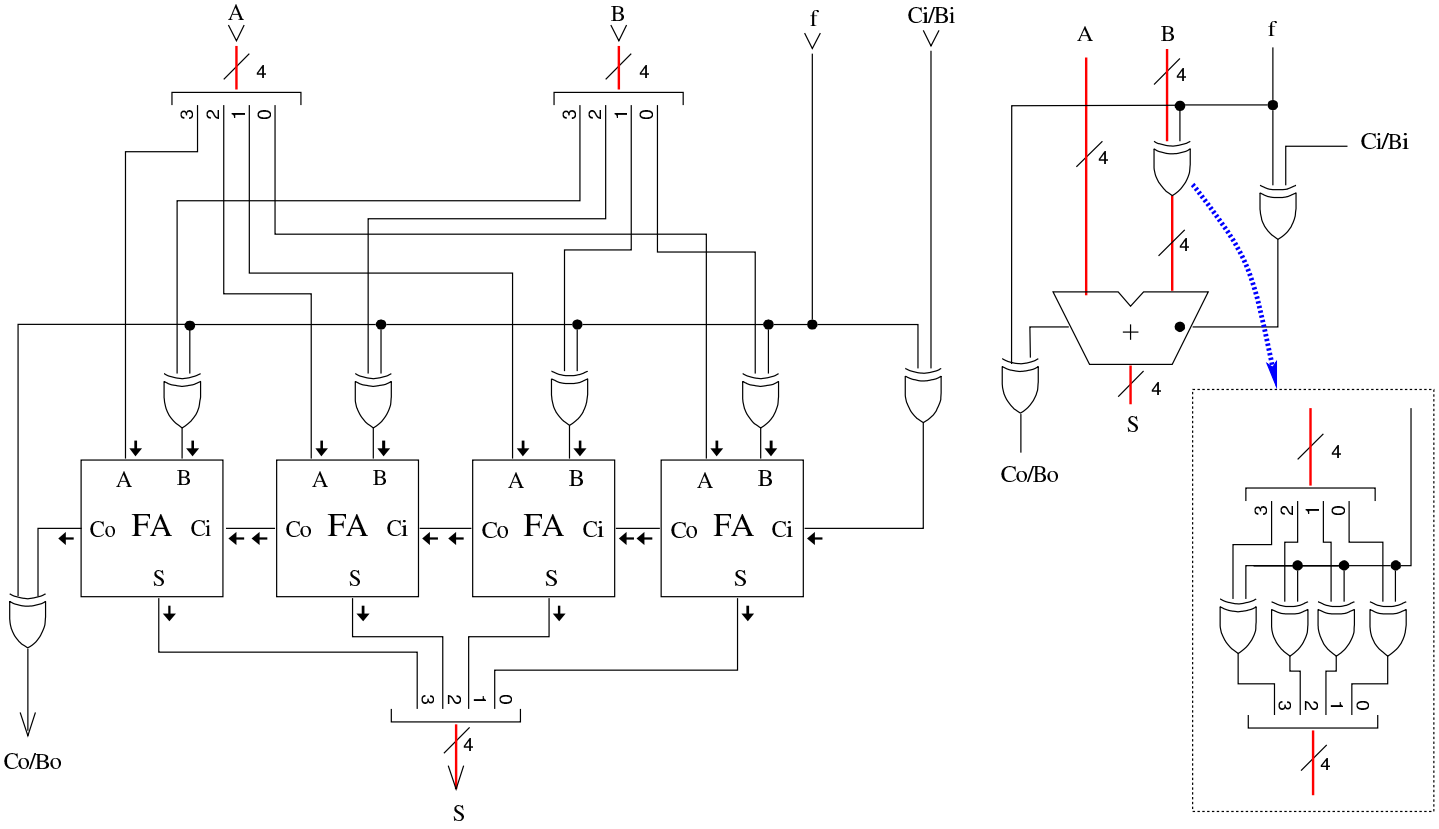
Figure 3.76. 4-digit subtractor: in that case it is calculated $\boldsymbol{A}-\boldsymbol{B}$. On the right side it is shown a compact representation of the same thing.



The addition and subtraction can be combined with XOR gates, instead of inverters,

with an additional function control input, which specify the action to take.

Figure 3.77. 4-digit addition or subtraction.



Listing 3.78. Gate level implementation of a 4-bit adder and subtractor.

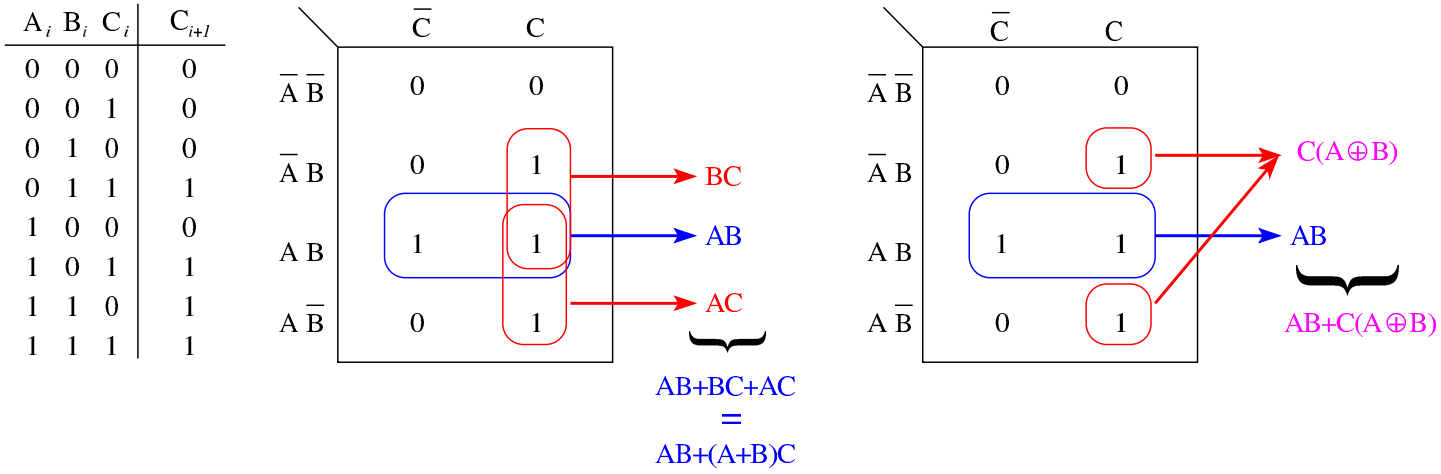
```
module addsub_4 (CoBo, S, f, A, B, CiBi);
  input [3:0] A, B;
  input f, CiBi;
  output CoBo;
  output [3:0] S;
  wire [4:0] C;
  //
  function add (input A, input B, input C);
    add = A^B^C;
  endfunction
  //
  function carry (input A, input B, input C);
    carry = A&B|A&C|B&C;
  endfunction
  //
```

```
   assign #0 C[0] = CiBi^f;
   assign #6 C[1] = carry (A[0], B[0]^f, C[0]);
   assign #6 C[2] = carry (A[1], B[1]^f, C[1]);
   assign #6 C[3] = carry (A[2], B[2]^f, C[2]);
   assign #6 C[4] = carry (A[3], B[3]^f, C[3]);
   //
   assign #5 S[0] = add (A[0], B[0]^f, C[0]);
   assign #5 S[1] = add (A[1], B[1]^f, C[1]);
   assign #5 S[2] = add (A[2], B[2]^f, C[2]);
   assign #5 S[3] = add (A[3], B[3]^f, C[3]);
   //
   assign #0 CoBo = C[4]^f;
endmodule
```

3.12 Carry lookahead

The addition with a significant quantity of bits, using the ripple-carry method of carry propagation is very slow. To accelerate the operation, every digit requires to get a valid previous carry in the shortest possible time, with the method known as ***carry-lookahead***. In the following expressions, the variables A_i, B_i and C_i represent the two addends and the input carry of the stage i (the digit taken into consideration); therefore, the carry generated from this stage is represented by the variable C_{i+1}.

Figure 3.79. Alternative ways to calculate the output carry.



The figure above shows that the carry can be synthesized in various ways, one of

which uses the XOR operator. Two equivalent expressions are chosen from the above picture:

$$C_{i+1} = A_iB_i+(A_i+B_i)C_i = A_iB_i+(A_i \oplus B_i)C_i$$

From these expressions, two new variables are defined, G_i and P_i, which stand for ***generation*** and ***propagation***, so that the carry C_{i+1} is defined by the value of G_i, P_i and C_i.

$$C_{i+1} = G_i+P_iC_i$$

It is obvious that G_i is equivalent to A_iB_i, whereas P_i can be considered either equal to A_i+B_i or to $A_i \oplus B_i$. If the first generated carry (C_1) can be obtained as $C_1=G_0+P_0C_0$, the second one is obtained as $C_2=G_1+P_1(G_0+C_0P_0)$, and the same way all the subsequent ones can be determined. The solutions for the first four carries follow:

$$C_1 = G_0+P_0C_0$$

$$C_2 = G_1+P_1C_1$$
$$C_2 = G_1+P_1(G_0+P_0C_0)$$
$$C_2 = G_1+P_1G_0+P_1P_0C_0$$
$$C_3 = G_2+P_2C_2$$
$$C_3 = G_2+P_2(G_1+P_1G_0+P_1P_0C_0)$$
$$C_3 = G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0$$
$$C_4 = G_3+P_3C_3$$
$$C_4 = G_3+P_3(G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0)$$
$$C_4 = G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0$$

To simplify the expressions, P_n and G_n are defined in the following way:

$$P_n = P_{n-1}P_{n-2}P_{n-1}\cdots P_1P_0$$

$$G_n = G_{n-1}+P_{n-1}G_{n-2}+P_{n-2}P_{n-1}G_{n-3}+\cdots+P_{n-2}P_{n-1}\cdots P_1G_0$$

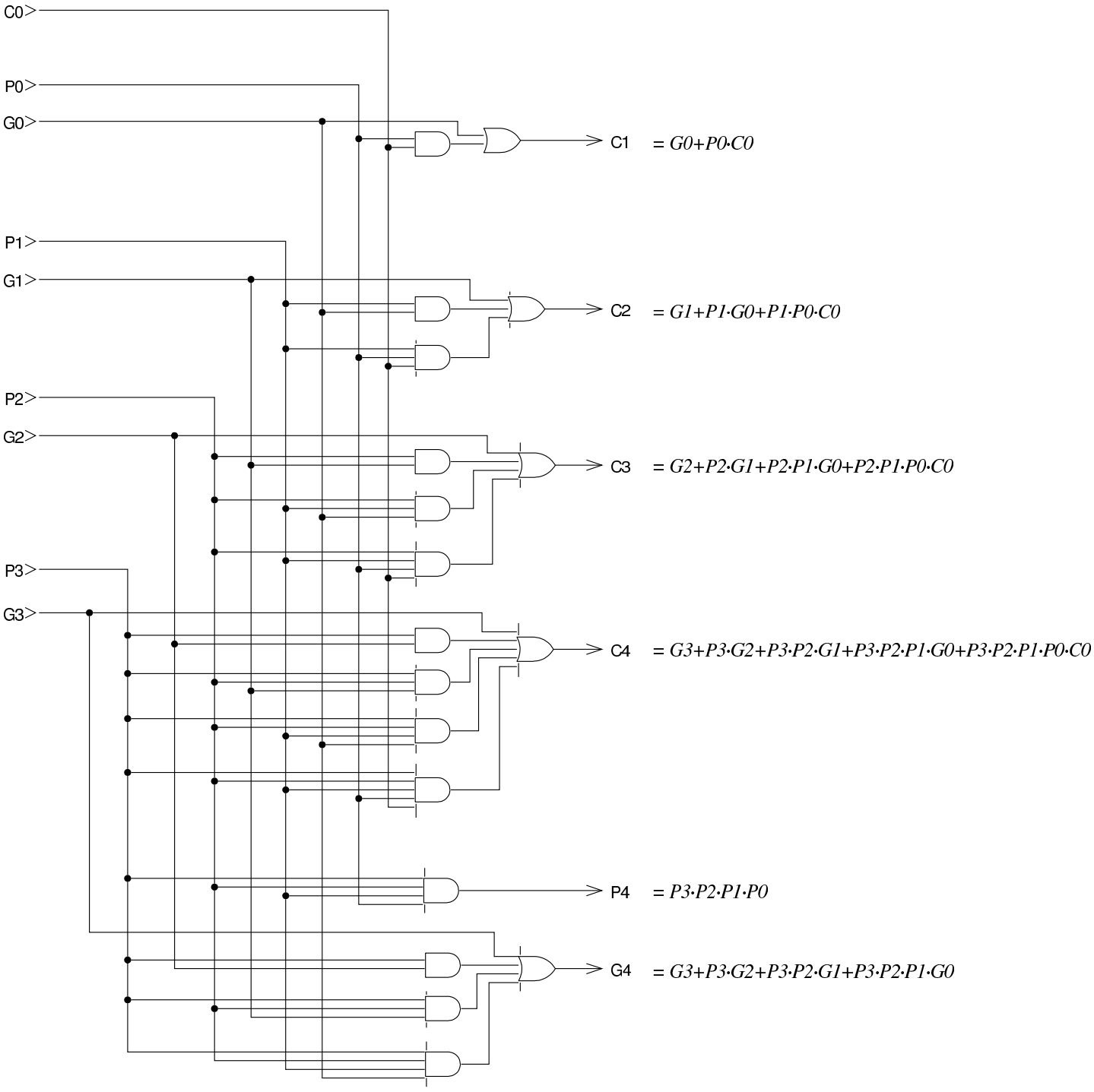
Having defined P_n and G_n, the carry C_n can be defined as:

$$C_n = G_n+P_nC_0$$

The following figure shows a combinational circuit that determines the carries of four binary digits, starting from the initial carry and the values from $B_{3..0}$ and $G_{3..0}$,

as described by the equations that defines this relation. The draw contains also the logic necessary to determine the value of B_4 and G_4, which can be useful to connect together more modules of this type.

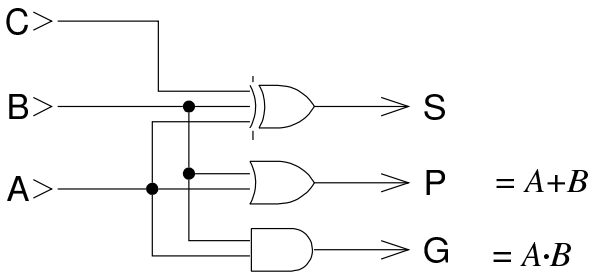
Figure 3.85. Diagram for the determination of four carries, starting from the initial one (C_0) and from the values of P_i and G_i, as described by the logic equations.

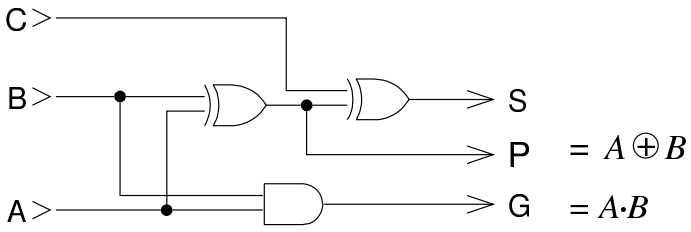


The adder that produces the values G and P can be synthesized without the output

carry, as this value is determined by the above module. There are two alternative solutions, because the $\boldsymbol{P}$ value can be defined either as $\boldsymbol{A_i}+\boldsymbol{B_i}$ or $\boldsymbol{A_i}\oplus\boldsymbol{B_i}$.

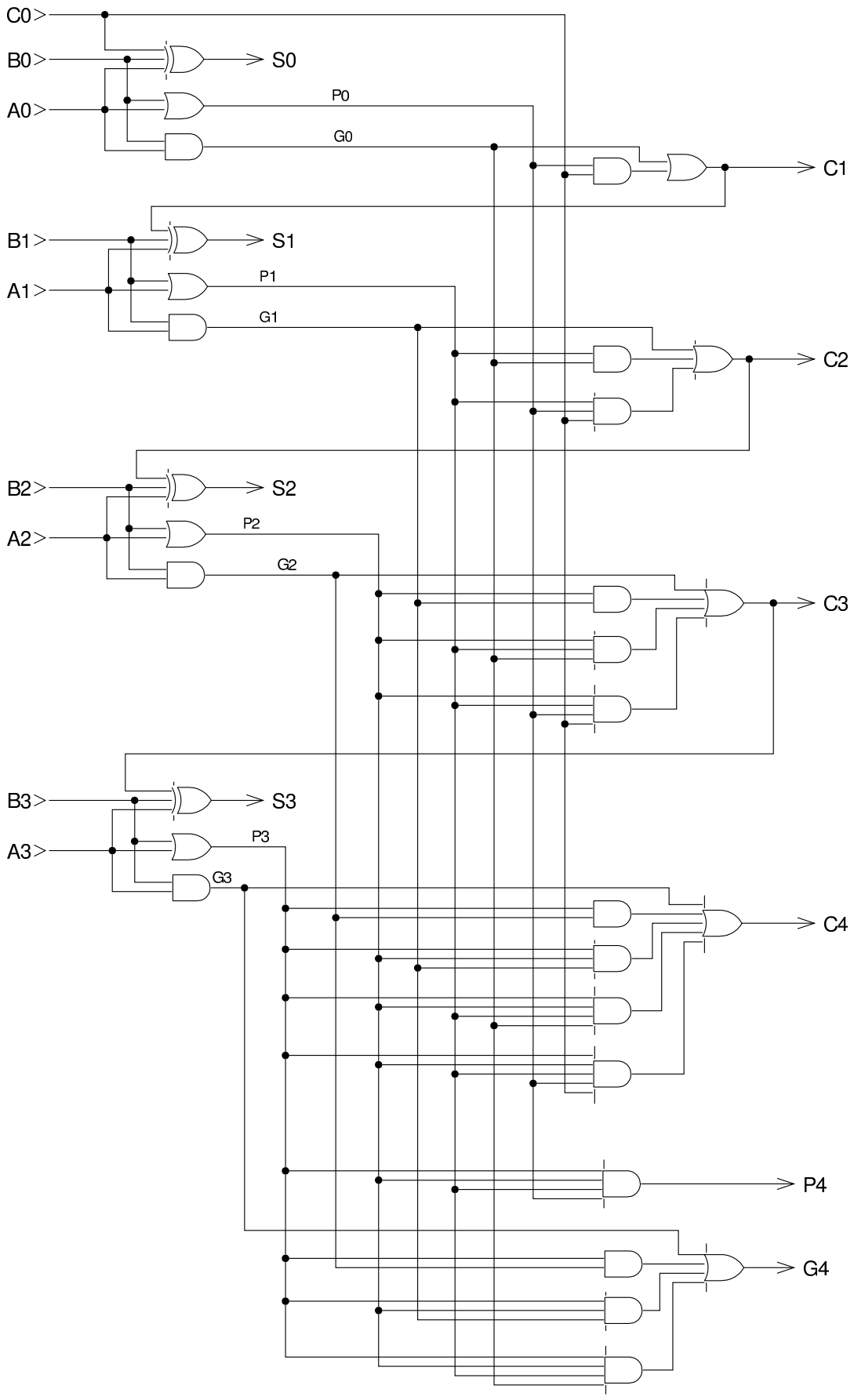
Figure 3.86. Addition modules with $\boldsymbol{P}$ and $\boldsymbol{G}$ outputs, but without the output carry.





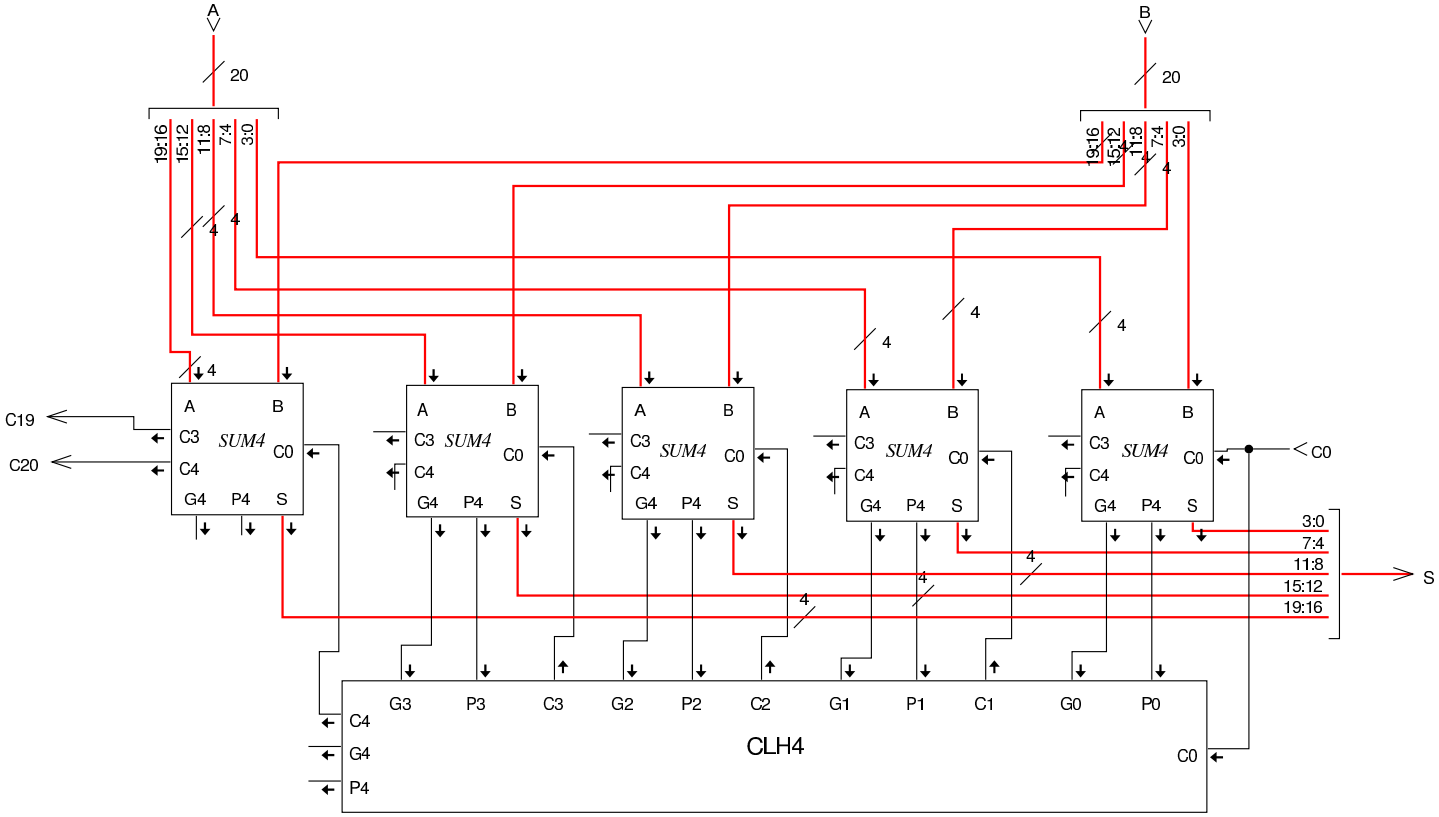
The following figure shows a 4-digit adder, made of the previous modules of addition and carry-lookahead.

Figure 3.87. Fast 4-digit adder.



An adder, like the one shown above, can be connected with other modules to increase the number of digits, without adding too much delay. In the following figure, the module SUM4 corresponds to the one described in figure 3.87, where only the last two carries are connected to the outside (the last two carries will be necessary to test if there is an overflow); the module CLH4 corresponds to the one described in figure 3.85, which is used to connect the addition modules.

Figure 3.88. 20-bit adder, made of 4-bit modules connected together.



Listing 3.89. Fast 4-digit adder, Verilog gate level implementation.

```
module SUM4 (C4, C3, P4, G4, S, A, B, C0);
  input [3:0] A, B;
  input C0;
  output C3, C4, P4, G4;
  output [3:0] S;
  wire [4:0] C, P, G;
  //
  assign #0 C[0] = C0;
  assign #5 S[0] = A[0]^B[0]^C[0];
  assign #5 S[1] = A[1]^B[1]^C[1];
```

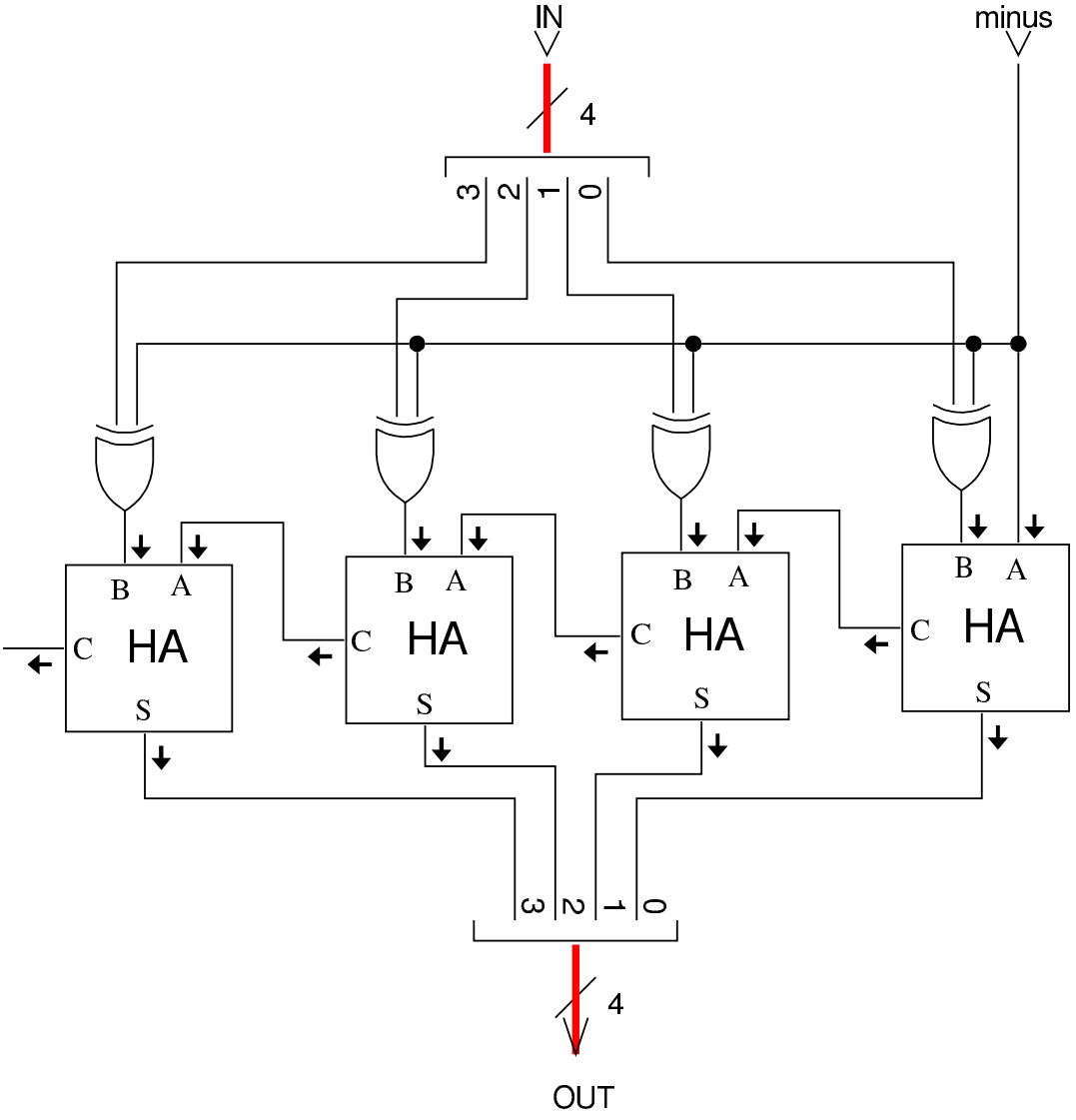
```
  assign #5 S[2] = A[2]^B[2]^C[2];
  assign #5 S[3] = A[3]^B[3]^C[3];
  //
  assign #5 P[0] = A[0]|B[0];
  assign #5 P[1] = A[1]|B[1];
  assign #5 P[2] = A[2]|B[2];
  assign #5 P[3] = A[3]|B[3];
  assign #6 P[4] = P[3]&P[2]&P[1]&P[0];
  //
  assign #5 G[0] = A[0]&B[0];
  assign #5 G[1] = A[1]&B[1];
  assign #5 G[2] = A[2]&B[2];
  assign #5 G[3] = A[3]&B[3];
  assign #7 G[4] = G[3]|P[3]&G[2]|P[3]&P[2]&G[1]
                  |P[3]&P[2]&P[1]&G[0];
  //
  assign #7 C[1] = G[0]|P[0]&C[0];
  assign #7 C[2] = G[1]|P[1]&G[0]|P[1]&P[0]&C[0];
  assign #7 C[3] = G[2]|P[2]&G[1]|P[2]&P[1]&G[0]
                  |P[2]&P[1]&P[0]&C[0];
  assign #7 C[4] = G[3]|P[3]&G[2]|P[3]&P[2]&G[1]
                  |P[3]&P[2]&P[1]&G[0]
                  |P[3]&P[2]&P[1]&P[0]&C[0];
  //
  assign #0 C4 = C[4];
  assign #0 C3 = C[3];
  assign #0 P4 = P[4];
  assign #0 G4 = G[4];
endmodule
```

A common 4-bit fast adder circuit is the 7483 (see i1.4.1).

3.13 2's complement

To change the sign of an integer binary number, it is necessary a circuit that calculates the 2's complement. To obtain the 2's complement it is required to calculate the 1's complement (bit inversion), adding 1.

Figure 3.90. 2's complement calculation, corresponding to the sign inversion of an 4-bit integer number.



The example in the above figure shows that the input data is inverted (negated) if the input ***minus*** is asserted, but the same ***minus*** value is added, producing the final 2's complement. If otherwise the ***minus*** input is not asserted, the input data is not changed and the output result remains the same.

3.14 Multiplication

The binary multiplication is a process that requires the addition and the bit shifting. To operate correctly on signed integers, the calculation should be done extending the input data to a double bit size; for example, if multiplicand and multiplier have only a 4-bit size, the result requires a size of up to 8 bits, but the calculation should be done as if also the input data were 8-bit wide. See the following examples.

Figure 3.91. signed and unsigned 4-bit multiplication.

integer multiplication without sign	*negative multiplicand, positive multiplier*	*positive multiplicand, negative multiplier*	*negative multiplicand, negative multiplier*
00001011×	11111011×	00000111×	11111011×
00001101=	00000101=	11111101=	11111101=
00001011+	11111011+	00000111+	11111011+
000000000+	000000000+	000000000+	000000000+
00101100+	11101100+	00011100+	11101100+
01011000+	00000000+	00111000+	11011000+
000000000+	000000000+	01110000+	10110000+
000000000+	000000000+	11100000+	01100000+
000000000+	000000000+	11000000+	11000000+
000000000=	000000000=	10000000=	10000000=
10001111	11100111	11101011	00001111

In the figure above, the first example on the left shows the multiplication of two unsigned values, which are extended using 0s on the most significant side, before the multiplication is done. The other examples are made with signed values and the negative ones are extended with 1s on the most significant side, to keep the original negative sign, before the operation take place.

To resolve the problem with a combinational circuit, it is necessary to cross the multiplicand and multiplier values, verifying at every useful position the coincidence of values equal to 1. The following two figures should be overlapped, as they show the concept in two phases: the outputs from the AND gates must be added vertically to generate the product.

Figure 3.92. Intersection between multiplicand and multiplier, using AND gates.

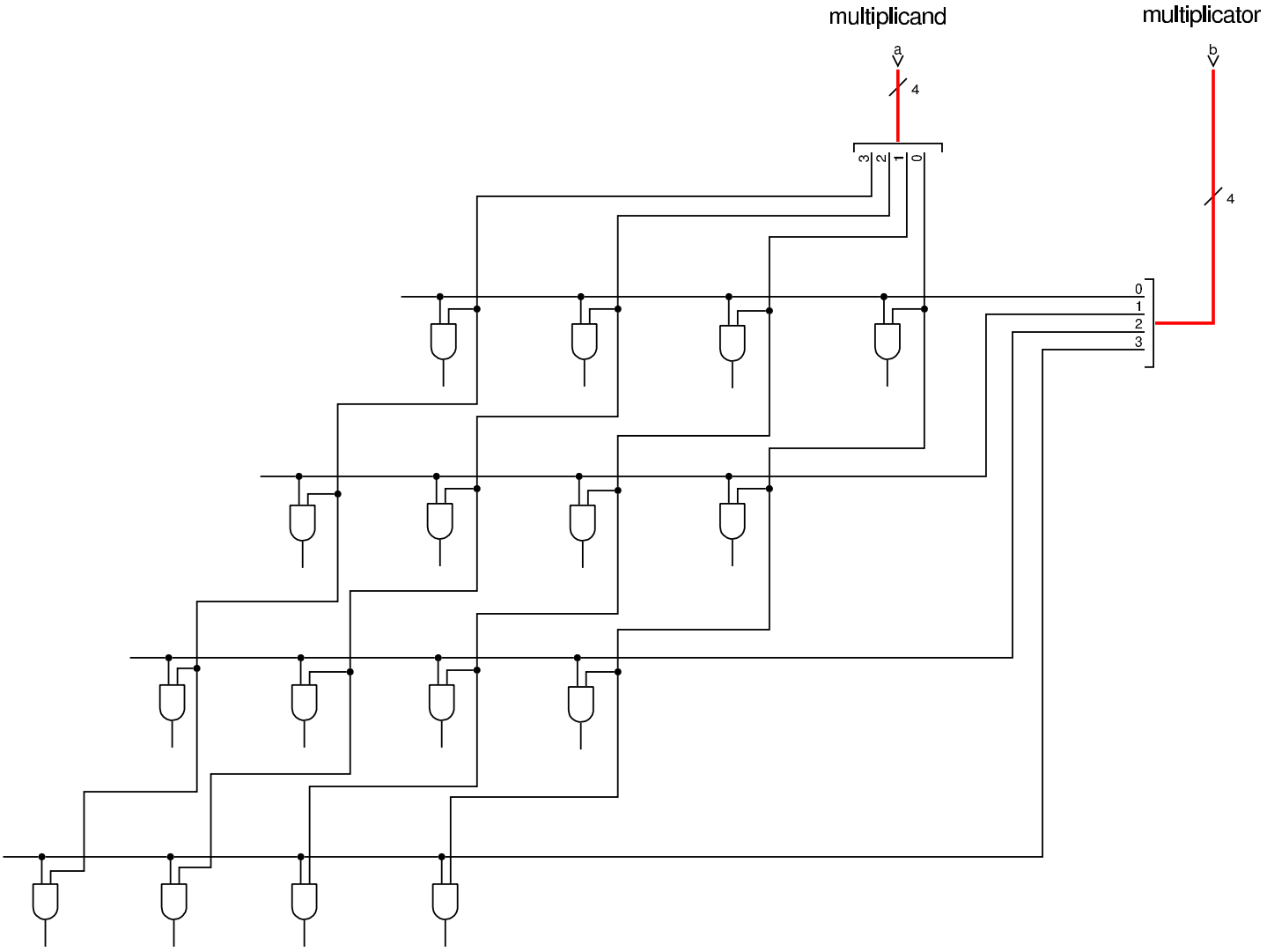
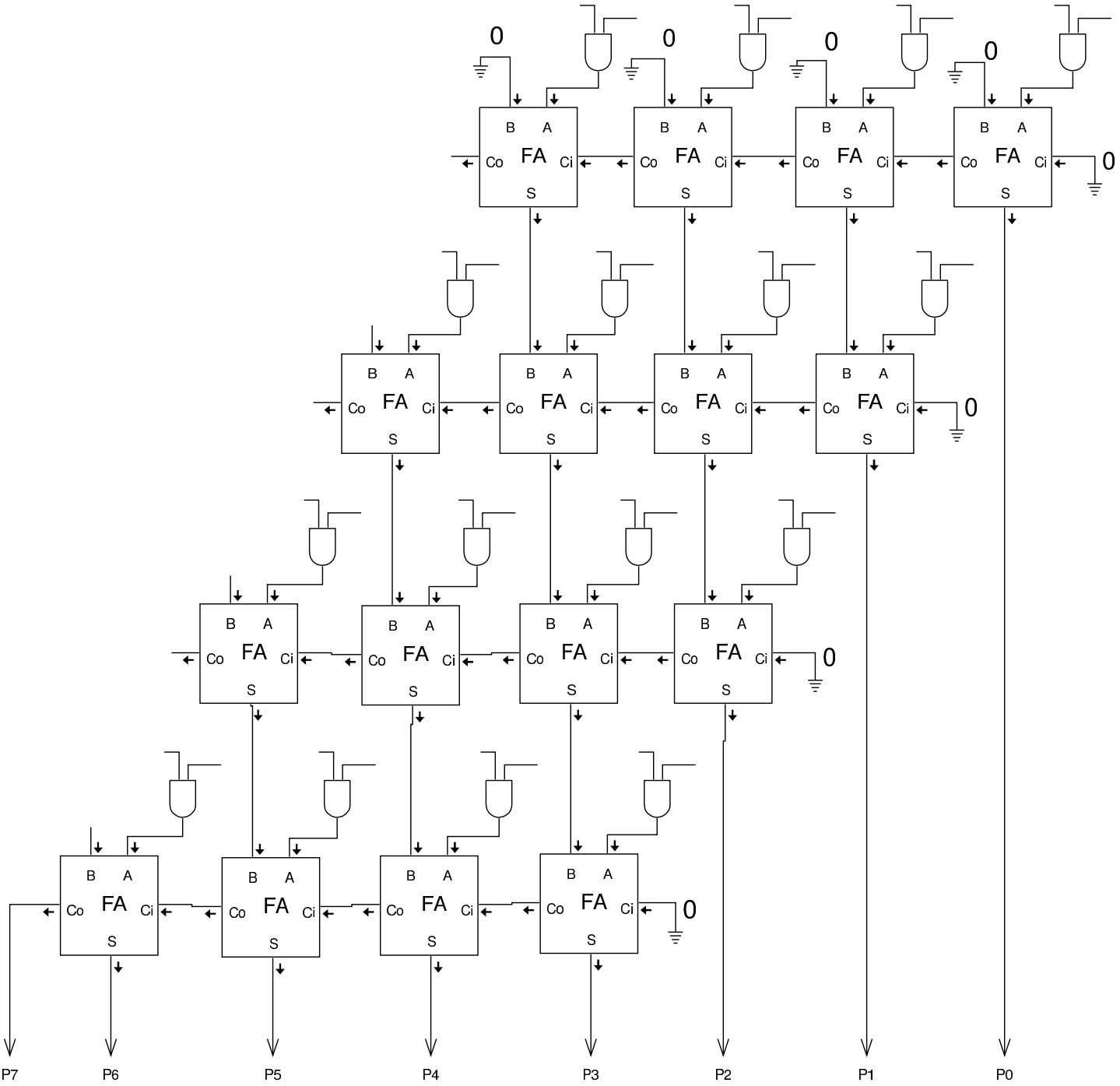
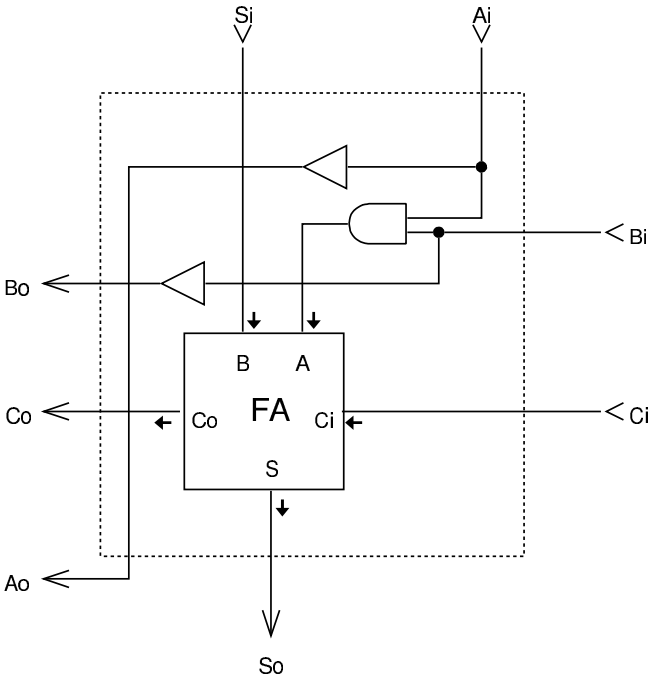


Figure 3.93. The value generated by the AND gates is added with full-adders.



To solve the problem in a nice way, it is necessary to build specific cells, using a full-adder. As it can be seen in the following figure, one addend receives the product calculated by the AND gate, whereas the other addend receives the value from the previous row.

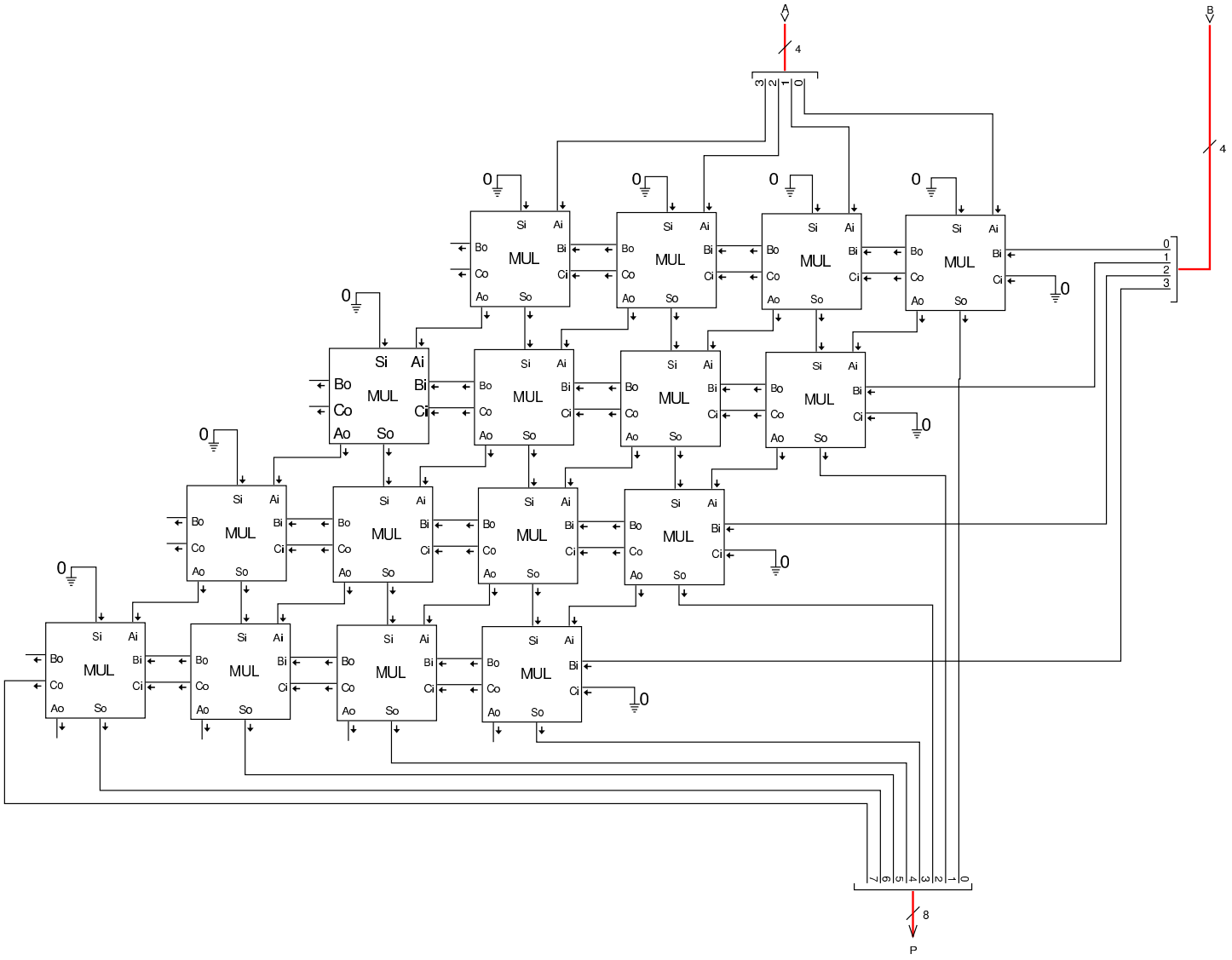
Figure 3.94. Cell used to build the matrix that adds and shifts the multiplicand.



Si = sum input
Ai = A input = multiplicand
Bi = B input = multiplicator
Ci = carry in
So = sum output = ((Ai AND Bi) + Si + Ci)
Ao = Ai
Co = carry out
Bo = Bi

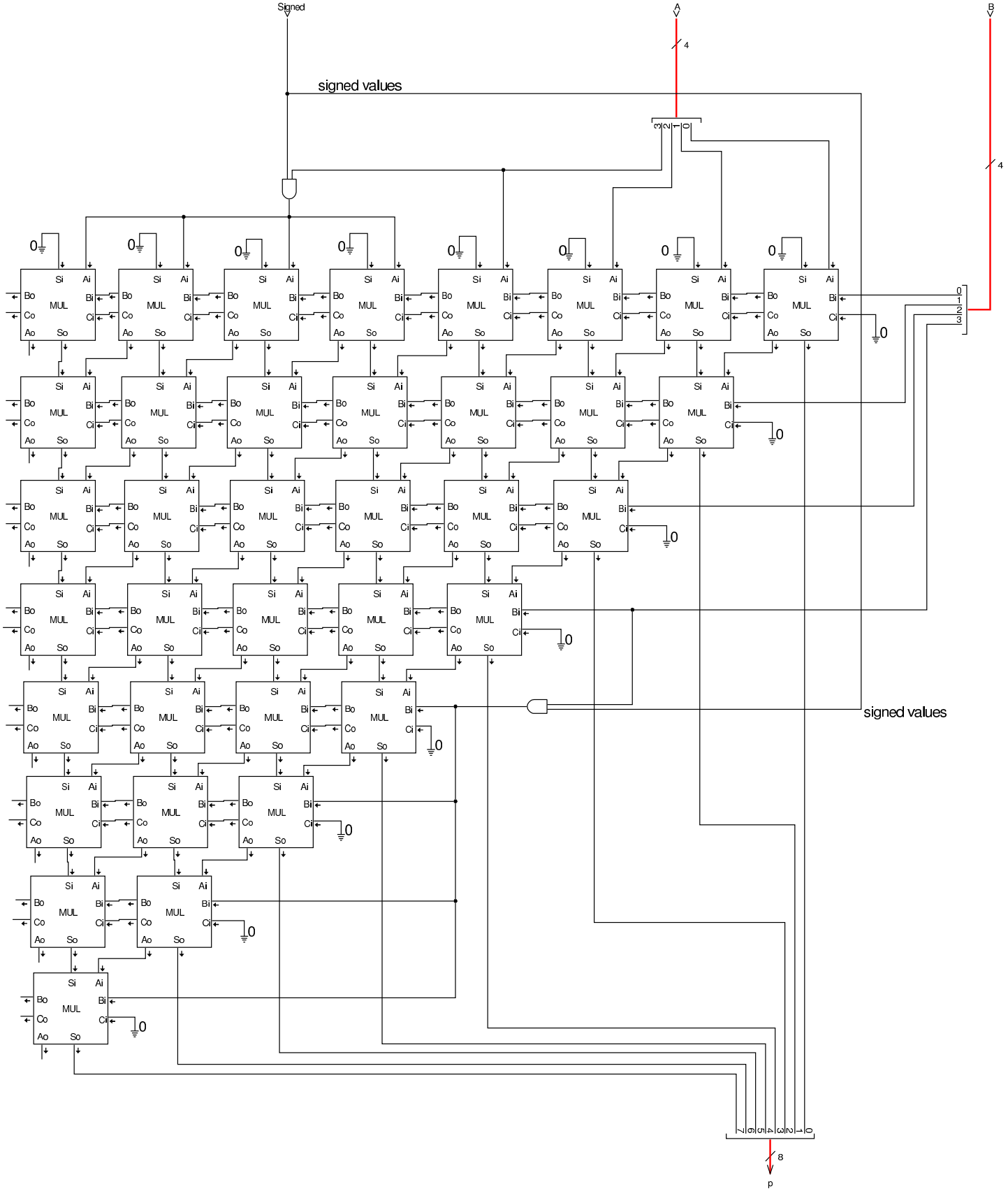
The cells of the previous figure should be connected properly together: the following figure shows a solution limited to unsigned multiplicand and multiplier.

Figure 3.95. 4-digit **unsigned** integer multiplication, producing an 8-digit result: $P=A\times B$.



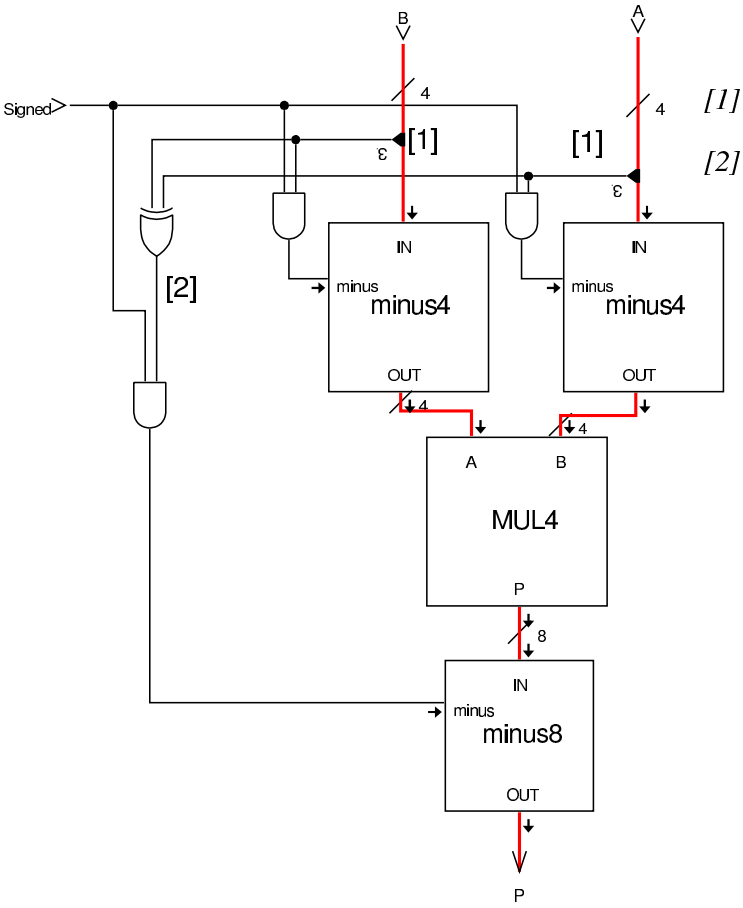
To multiply signed integers, it is necessary an additional control input, used to specify if the input data is signed, then the net should be extended as if the multiplicand and the multiplicator have a double bit size, extending their values conforming to the original sign that they are intended to have.

Figure 3.96. Full solution (but excessively heavy) to calculate the product of two 4-bit integers.



The sign problem can be resolved changing the sign, before and after the multiplication, so that a simple unsigned circuit can be used. In the following figure, the module ***MUL4*** is a multiplication circuit that can accept only unsigned values; the modules ***minus4*** and ***minus8*** are circuits that can change the sign of the values that they filter, if their input ***minus*** is asserted.

Figure 3.97. Signed integer multiplication, using an adapted unsigned multiplier.



[1] the most significant bit is read to know the sign of the integer

[2] multiplicand and multiplier have different signs

The figure above shows that from the inputs ***A*** and ***B*** the most significant bit is used to know their signedness; if they are negative, their value is converted into positive, before the multiplication. After the multiplication, if the input signs were different, the result is converted into a negative value. This sign transformation is enabled by the control input ***Signed***, so that unsigned number can be multiplied as well. The module ***minus4*** corresponds to the one shown in the section 3.13, whereas the ***minus8*** module is an extention working with 8 bits.

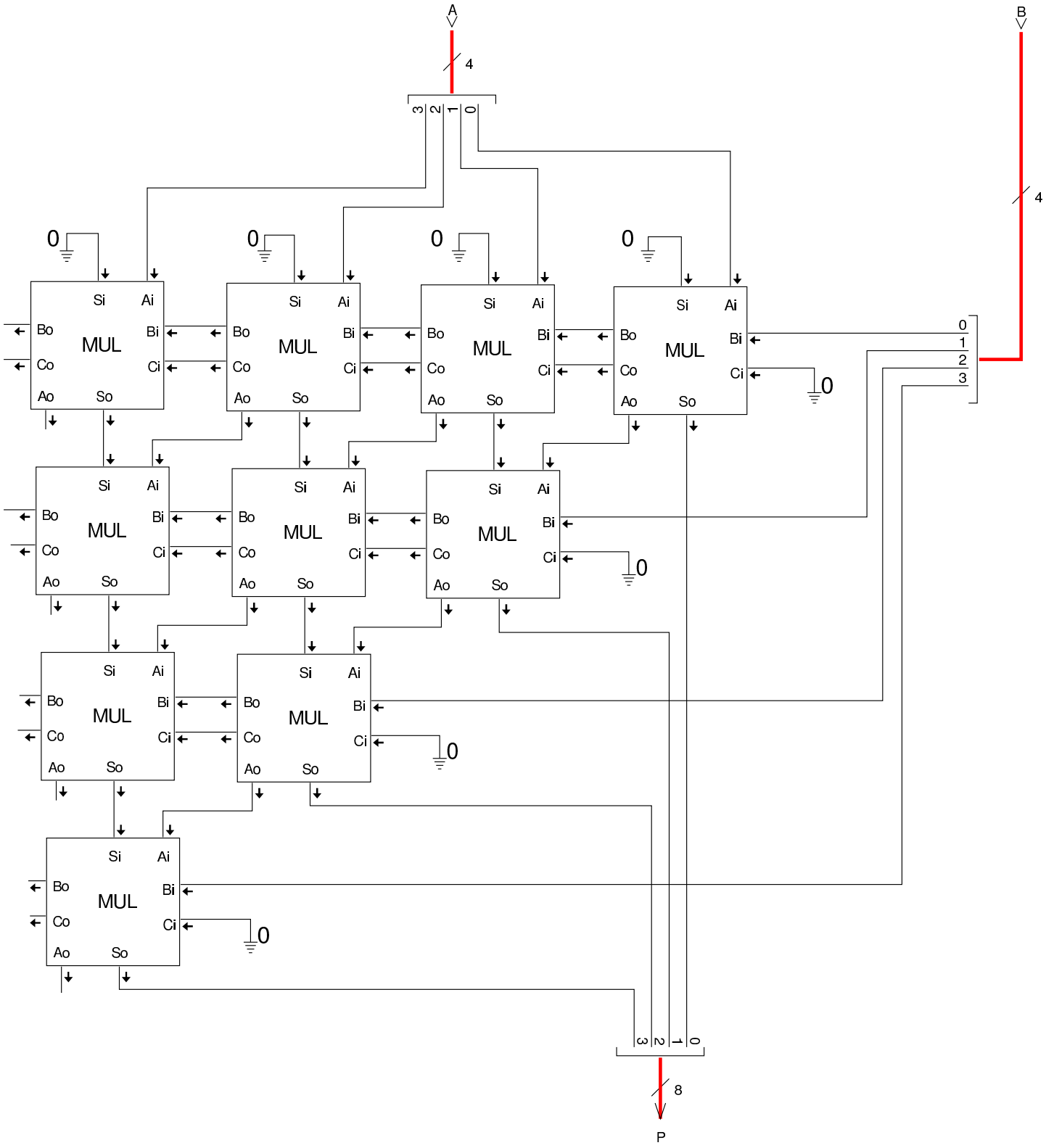
It should be observed that if the result is truncated to the same number of digits than the multiplicand and the multiplier, assuming that the result can fit in it and that the input data is always signed, then the result is valid, without the need to handle the

signedness. The following examples try to demonstrate this assertion.

Figure 3.98. 4-digit binary multiplication, truncating the result at the same size as multiplicand and multiplier.

integer multiplication without sign	*negative multiplicand, positive multiplier*	*positive multiplicand, negative multiplier*	*negative multiplicand, negative multiplier*
1 0 1 1 ×	**1 1 0 1** ×	**0 1 0 1** ×	**1 0 1 1** ×
1 1 0 1 =	**0 0 1 0** =	**1 1 1 0** =	**1 1 0 1** =
1 0 1 1 +	**0 0 0 0** +	**0 0 0 0** +	**1 0 1 1** +
0 0 0 0 +	**1 0 1** 0 +	**1 0 1** 0 +	**0 0 0** 0 +
1 1 0 0 +	**0 0** 0 0 +	**0 1** 0 0 +	**1 1** 0 0 +
1 0 0 0 =	**0** 0 0 0 =	**1** 0 0 0 =	**1** 0 0 0 =
1 1 1 1	1 0 1 0	0 1 1 0	1 1 1 1
invalid result because it is truncated too early	*valid result*	*valid result*	*valid result*

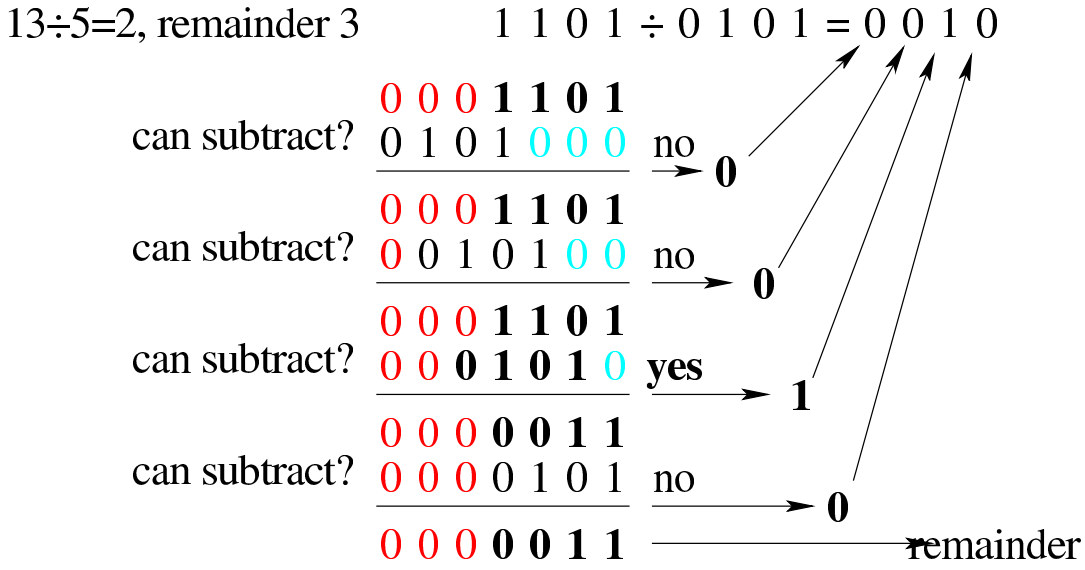
Figure 3.99. Simplified multiplication combinational circuit with the result truncated to the same bit size as the multiplicand and multiplier.



3.15 Division

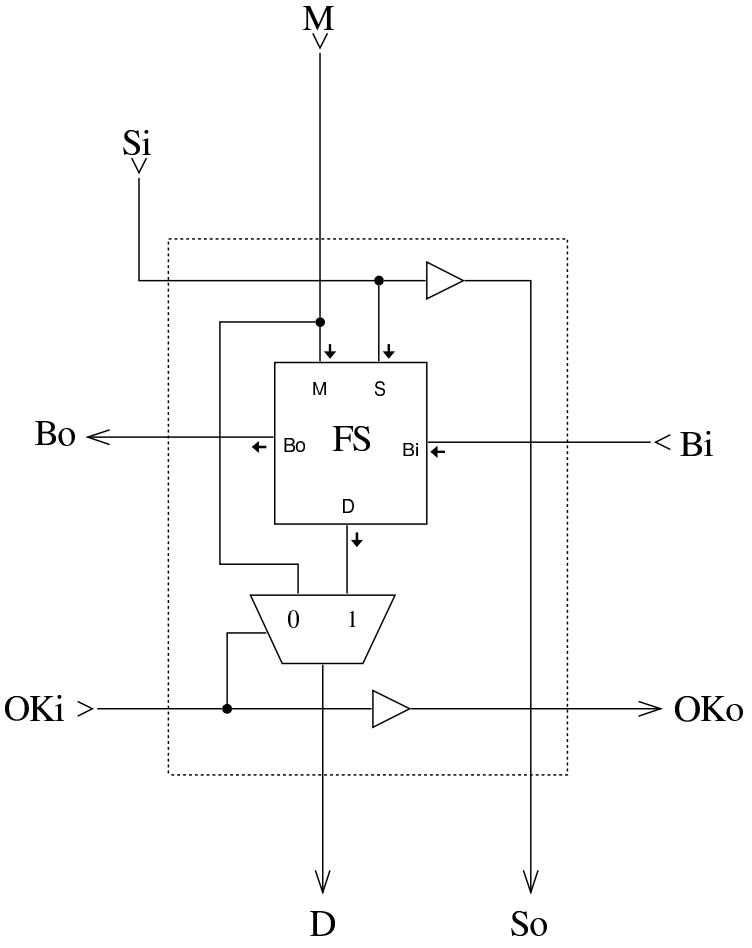
The binary division requires to iterate the subtraction and the bit shifting, as the following example shows: the divisor is compared with the dividend, starting from the most significant digits, subtracting the divisor from the dividend only if it is possible. That way, the integer division result (quotient) is made of 1s digits when the subtraction is possible and 0s otherwise, whereas the remainder is what remains after all the subtractions.

Figure 3.100. Unsigned integer division procedure.



To resolve the division problem with a combinational circuit the full-subtractor module can be used, including some more circuitry to check if the subtraction can be applied. As for the multiplication, a specific module can be designed.

Figure 3.101. Conditional subtraction cell.



SF = full−subtractor

M = minuend

Si = input subrahend

So = output subtrahend: copy for the next cell

Bi = Borrow in

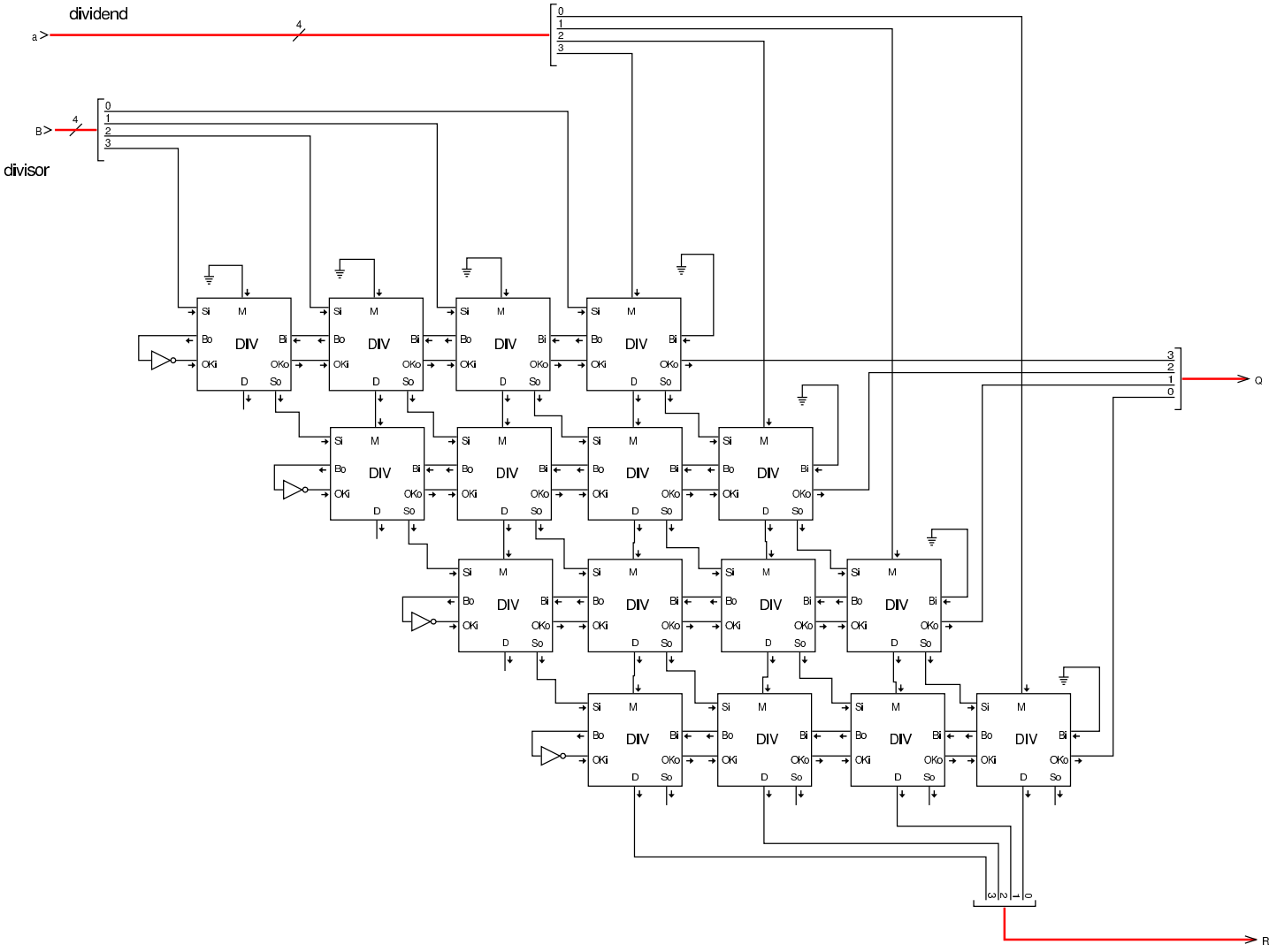
Bo = borrow out

OKi = OK in: the subtraction can be done,

OKo = OK out: copy of OKi for the next cell

With these cells, after every subtraction it is known if there is a borrow from the last ***Bo*** output (*borrow out*) on the left. If there is a borrow, then the subtraction can not be done, so the final borrow output is used inverted to select the subtracted value or the original one (the ***OKi*** input).

Figure 3.102. Solution example for the unsigned integer division.

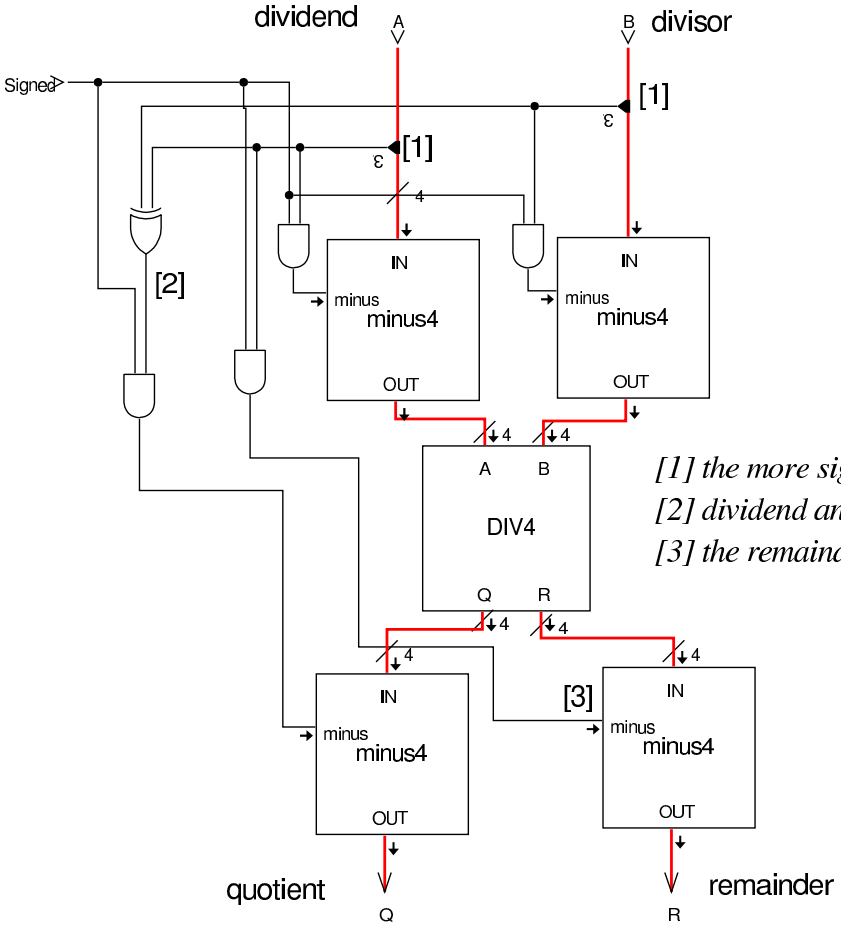


When the division is signed, the following cases should be considered to decide the quotient and the remainder signs:

Dividend	Divisor	Quotient	Remainder
+	+	+	+
+	−	−	+
−	+	−	−
−	−	+	−

To solve the signed division problem, a mechanism that changes the signs, before and after the division, might be the best way.

Figure 3.104. Signed integer division: ***DIV4*** can divide only unsigned numbers; ***minus4*** changes the sign of a value if the input ***Signed*** is asserted.

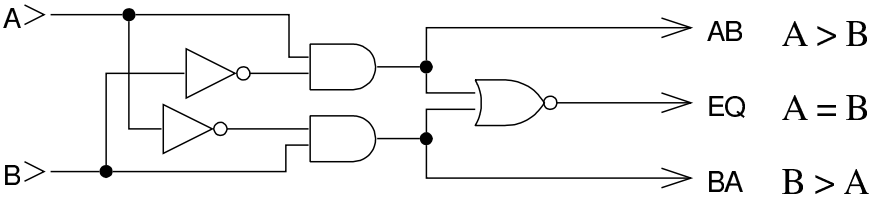


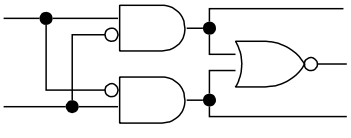
[1] the more significant bit is used to know the sign of the value
[2] dividend and divisor signs are different
[3] the remainder sign depends only from the dividend

3.16 Magnitude comparator

To do a magnitude comparator with two unsigned binary integers, it is necessary to compare the digits of both values, starting from the more significant ones. For every digit comparation it is necessary to establish which is the higher or if they are equal. The following figure shows the comparation between two simple logic values.

Figure 3.105. 1-digit magnitude comparator; the diagram on the right side is the same circuit with a more compact representation.

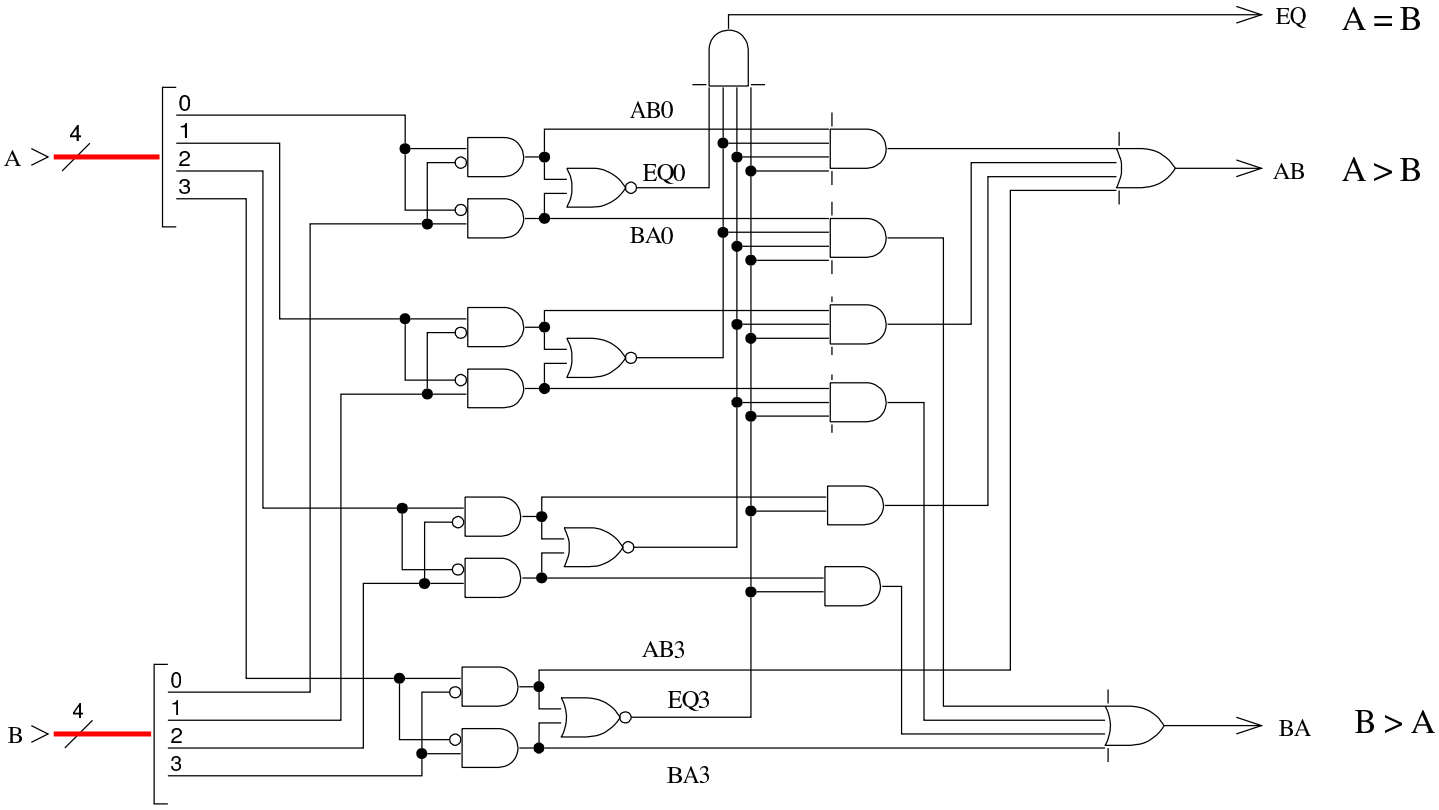




When more bits are compared, the match starts from the most significant bit, which prevails over the other. If the most significant bits are the same, the next digit should

be checked to find which value is greater than the other; if all the digits of the two values are the same, the values are equal.

Figure 3.106. 4-bit unsigned integer magnitude comparator.



Listing 3.107. 4-digit magnitude comparator, Verilog gate level implementation.

```
module magnitude_4 (BA, EQ, AB, A, B);
  input [3:0] A, B;
  output BA, EQ, AB;
  wire [3:0] ab;
  wire [3:0] eq;
  wire [3:0] ba;
  //
  assign #3 ab[0] = A[0]&~B[0];
  assign #3 ab[1] = A[1]&~B[1];
  assign #3 ab[2] = A[2]&~B[2];
  assign #3 ab[3] = A[3]&~B[3];
  //
  assign #3 ba[0] = ~A[0]&B[0];
  assign #3 ba[1] = ~A[1]&B[1];
```

```
  assign #3 ba[2] = ~A[2]&B[2];
  assign #3 ba[3] = ~A[3]&B[3];
  //
  assign #3 eq[0] = ~(ab[0]|ba[0]);
  assign #3 eq[1] = ~(ab[1]|ba[1]);
  assign #3 eq[2] = ~(ab[2]|ba[2]);
  assign #3 eq[3] = ~(ab[3]|ba[3]);
  //
  assign #4 EQ = eq[0]&eq[1]&eq[2]&eq[3];
  //
  assign #3 AB = (ab[0]&eq[1]&eq[2]&eq[3])
               |(ab[1]&eq[2]&eq[3])
               |(ab[2]&eq[3])
               |ab[3];
  //
  assign #3 BA = (ba[0]&eq[1]&eq[2]&eq[3])
               |(ba[1]&eq[2]&eq[3])
               |(ba[2]&eq[3])
               |ba[3];
endmodule
```

A common 4-bit magnitude comparator circuit is the 7485 (see i1.4.2).

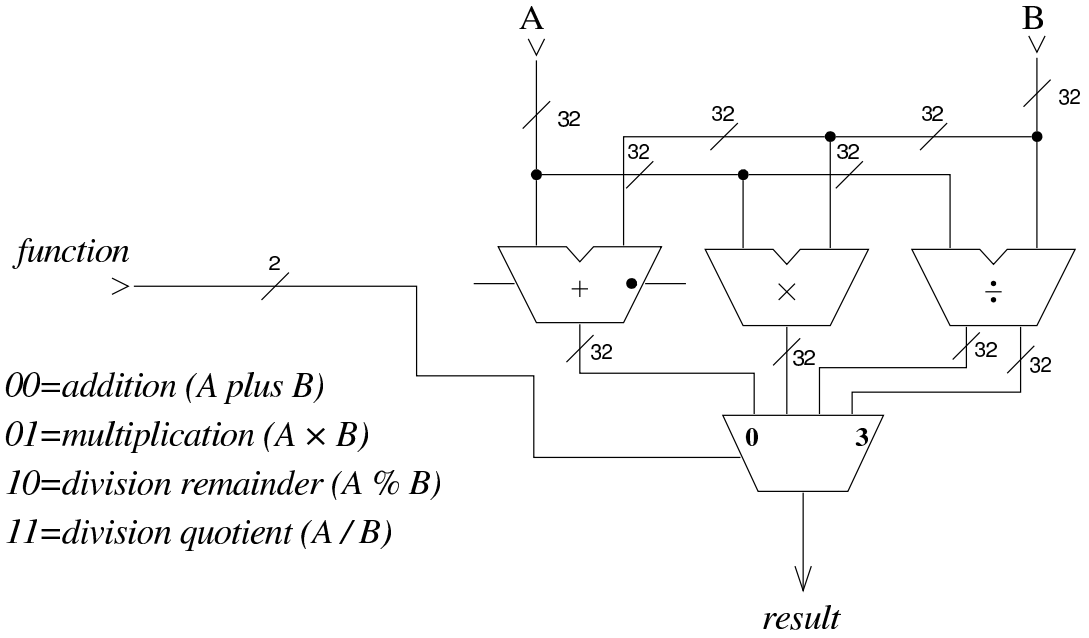
4.1 Status flags 150
4.2 Size truncation 152
4.3 Sign inversion 154
4.4 Addition and subtraction 156
4.5 Shift and rotation 158
4.6 Magnitude comparation 162
4.7 Multiplication 165
4.8 Division 171
4.9 Logic unit 176
4.10 Joining all the module together 177
4.11 Bit sliced ALU 179

addition 156 ALU 177 179 bit-size 152 bit slice 179 carry 150 comparation 162 division 171 flag 150 multiplication 165 negative 150 overflow 150 156 158 rank 150 152 rotation 158 shift 158 sign 150 154 size 152 status 150 subtraction 156 zero 150

The arithmetic-logic unit (ALU) is a combinational circuit that contains the main calculation functions of a CPU. It is usually composed by some distinguished circuits, everyone of which does a single or a little group of calculations. These components are usually joined together through a multiplexer that selects the chosen result.

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

Figure 4.1. Simplified diagram of three components working together in a single ALU: everyone receives a couple of input values, but the output is selected by a multiplexer, through a control code that represents the function requested to the ALU.



Where possible, the examples of the chapter are divided into 8-bit modules, which might be connected together to work with multiple bit sizes: 16, 32, 64.

Common circuit components used to build an ALU are: 7483, 7485, 74181, 74182, 74381 and 74382 (see u0.4).

4.1 Status flags

«

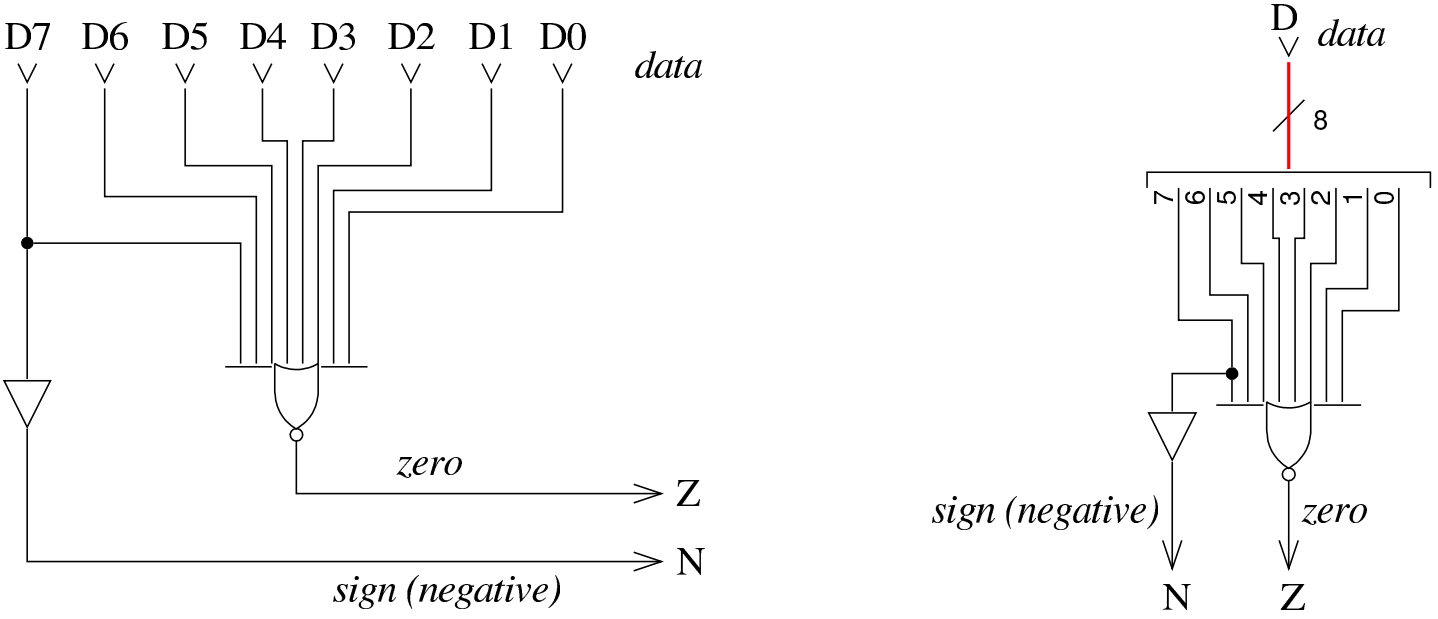
For some kind of operation the ALU needs to remember a previous status; for example, an addition might be done in more than one step, taking into account the previous carry. Some flags are used to keep track of the status of previous operations; the most common ones are: ***carry***, ***zero***, ***sign*** and ***overflow***.

Name	Common abbreviation	Description
zero	Z	Asserted when the last operation result was zero.
negative sign	N	Asserted when the last operation result seems to be negative, because the most significant bit is 1.

Name	Common abbreviation	Description
carry	C	Usually asserted when the last operation requires an additional digit to store the value and the effective result is truncated, but used also for other situations when a digit should be saved for iterated calculations.
overflow	O V	The overflow is a condition that arise when the result of an operation is not valid because it is truncated at the most significant side or because it has the wrong sign. When adding two integer values there is an overflow if the last two carries are different.

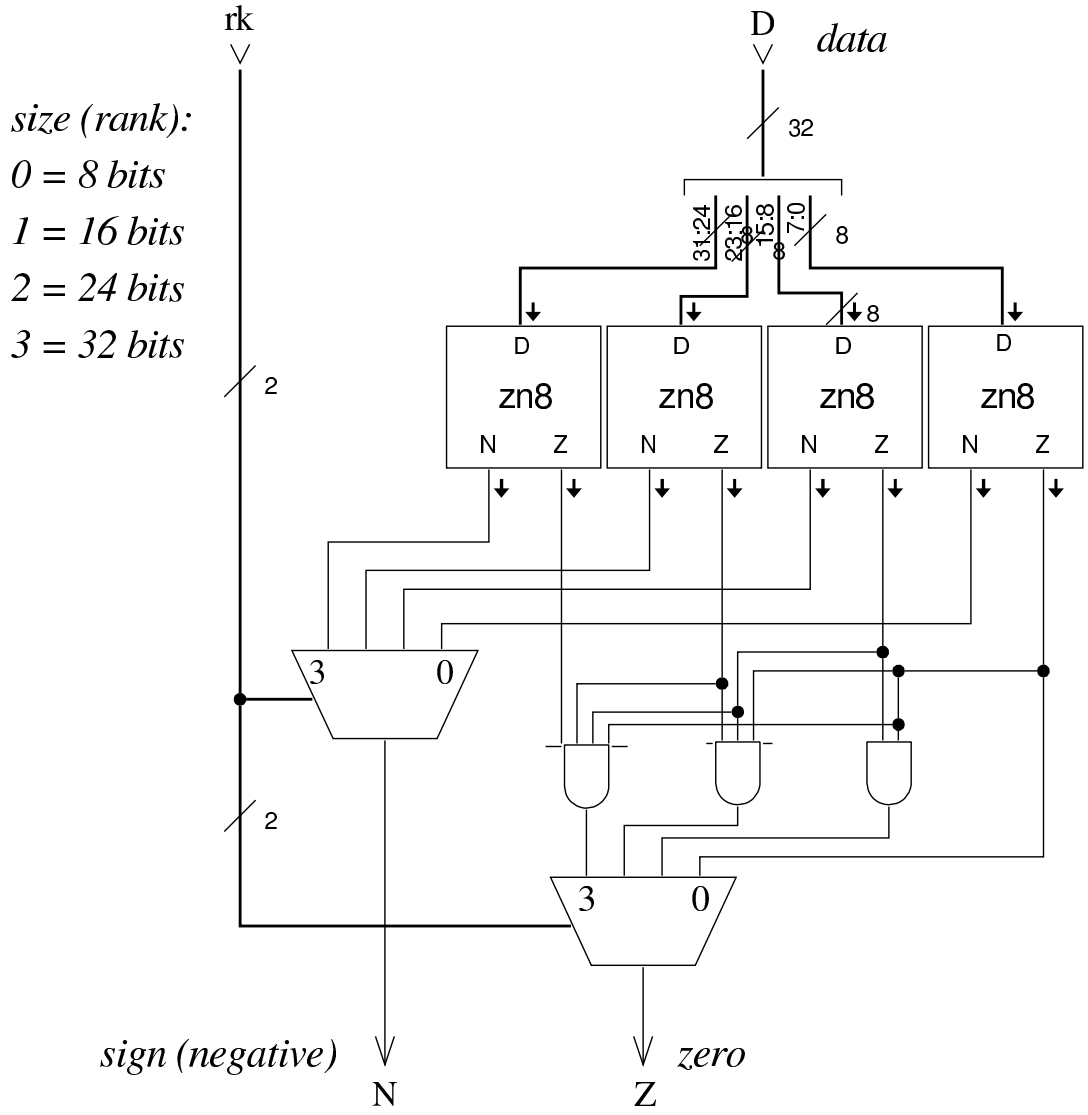
To verify a result equal to zero it is possible to check the output value with a NOR gate, which produces a 1 only if all the inputs are 0. It is also easy to find the sign of a value (assuming that the value is signed) checking the most significant bit.

Figure 4.3. Module **`zn8`**: check for zero and for a possible sign. On the right side the same circuit appears with a compact diagram, where the eight input lines are grouped together in a single multiline port.



If it is required to operate with different bit-sized data, the above module can be used in paralel, adding a ***rank*** selection, like the following figure shows.

Figure 4.4. Module **zn32**: zero and sign evaluation for integers from 8 to 32 bits. The input port ***rk***, which stands for 'rank', is a control function allowing to specify the desired bit-size: zero means 8 bits, up to 3 that means 32 bits. The module **zn8** is described in figure 4.3.



4.2 Size truncation

The modules described in the following sections allow to operate on 32-bit values and on smaller sizes (multiples of 8). When the specified size is smaller than 32 bits, the results are valid only inside that particular rank and there might be also more contents that should be ignored on the more significant side, beyond the specified range. The following module allows to filter a value and even to extend the sign, based on a selected size.

Figure 4.5. Module **`rk`**: filter and sign extension of a value, based on a requested size.

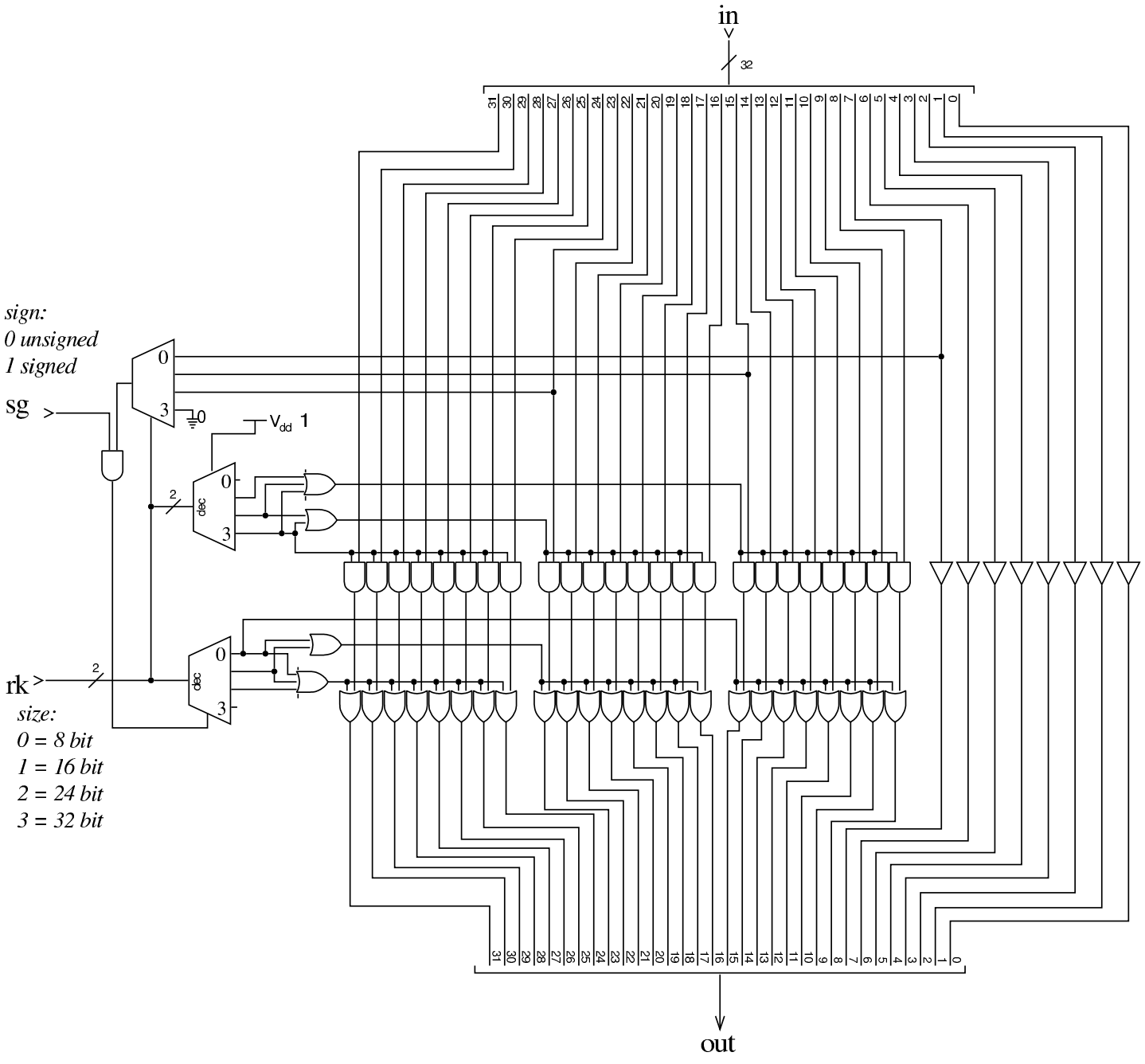
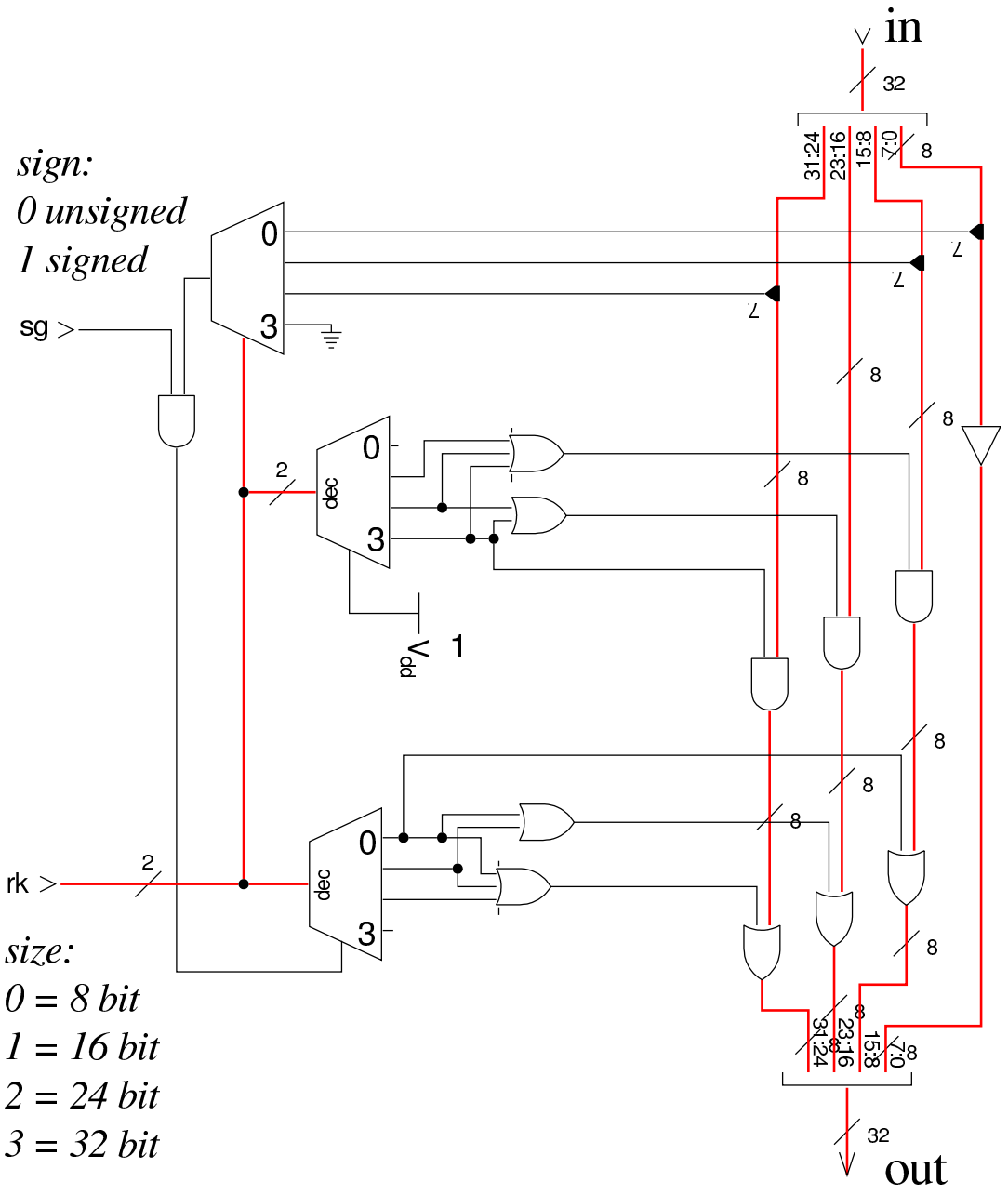


Figure 4.6. Module **rk**, identical to the previous picture, but designed in a more compact way.



4.3 Sign inversion

To assist the multiplication and the division it is important a module that can invert the sign of a (signed) binary integer number. The module **minus8** works on 8 bits and it is used to build other inversion modules with a bigger bit-size.

Figure 4.7. Module **`minus8`**: 8-bit sign inversion. If the inputs ***M*** and ***Ci*** are asserted the sign is inverted, otherwise the value is not changed. The module **`ha`** is a common half adder.

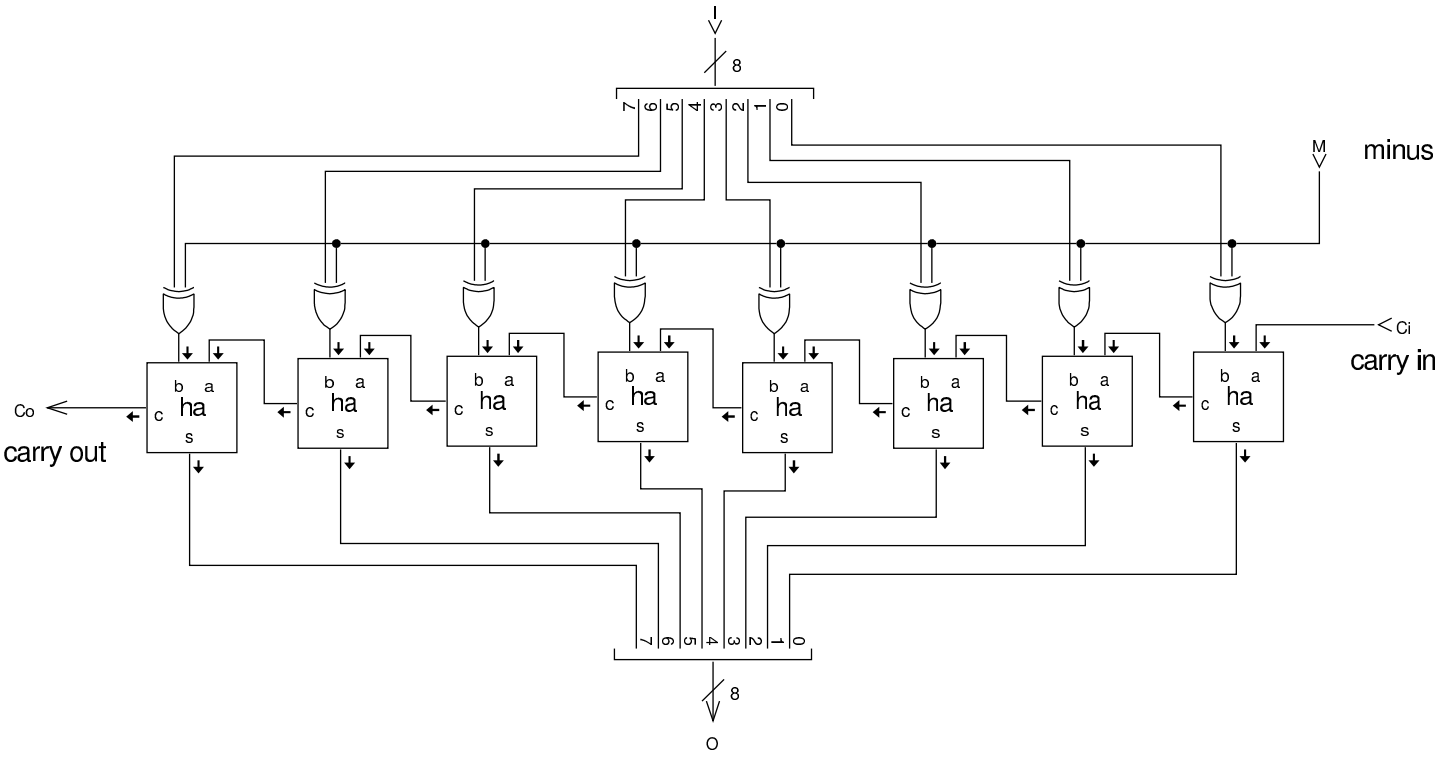
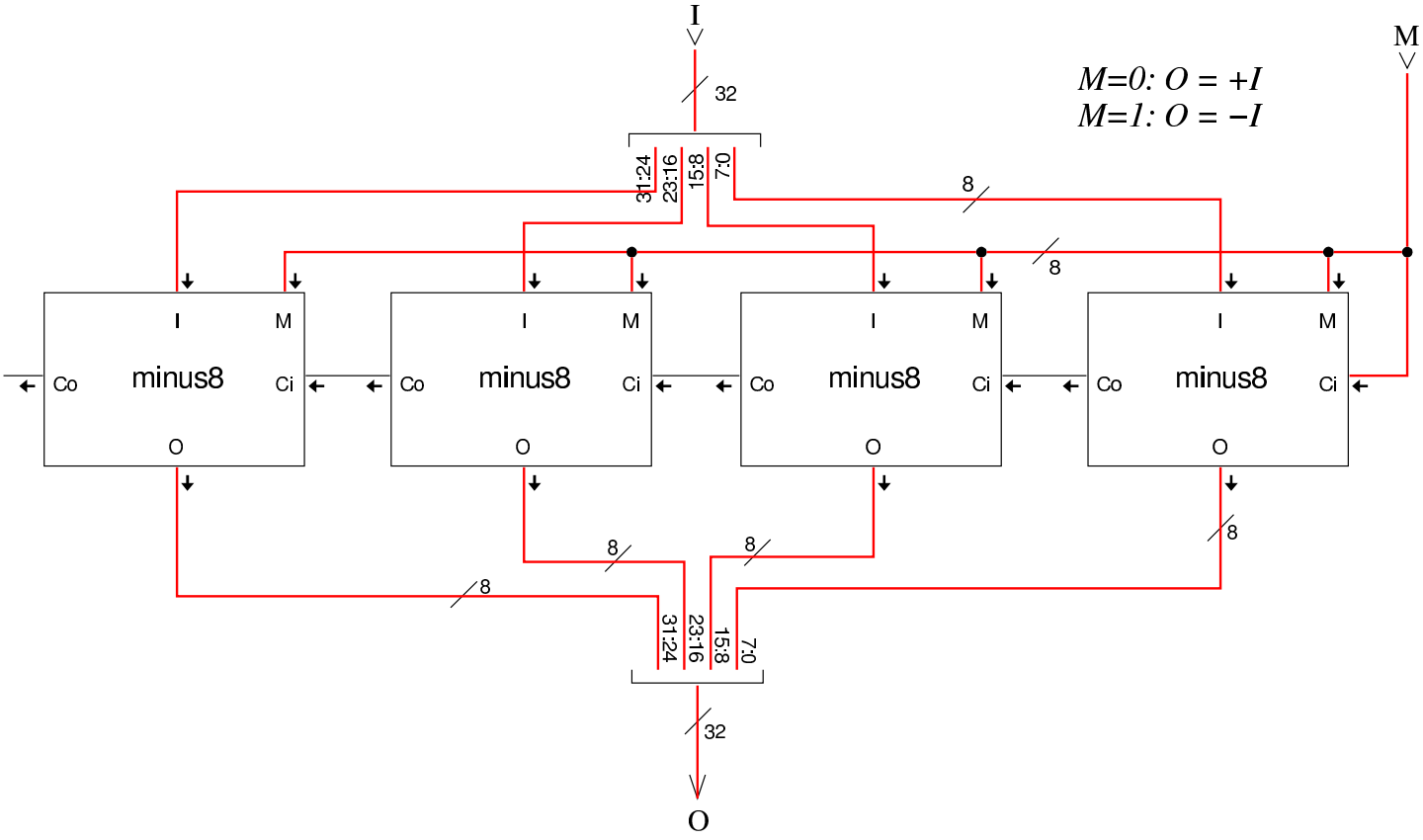


Figure 4.8. Module **`minus32`**: 32-bit sign inversion, made of four **`minus8`** modules, as described in figure 4.7.



4.4 Addition and subtraction

«

The following figure shows an 8-bit module that is able to add and subtract integer values, handling correctly either the carry or the borrow, signaling also the overflow (if the last two carries are different).

Figure 4.9. Module **`as8`**: addition and subtraction of 8-bit binary integers. The input ***f*** (function) selects the addition or the subtraction; the output ***O*** gives the overflow (if there is one). The module **`fa`** is a full-adder.

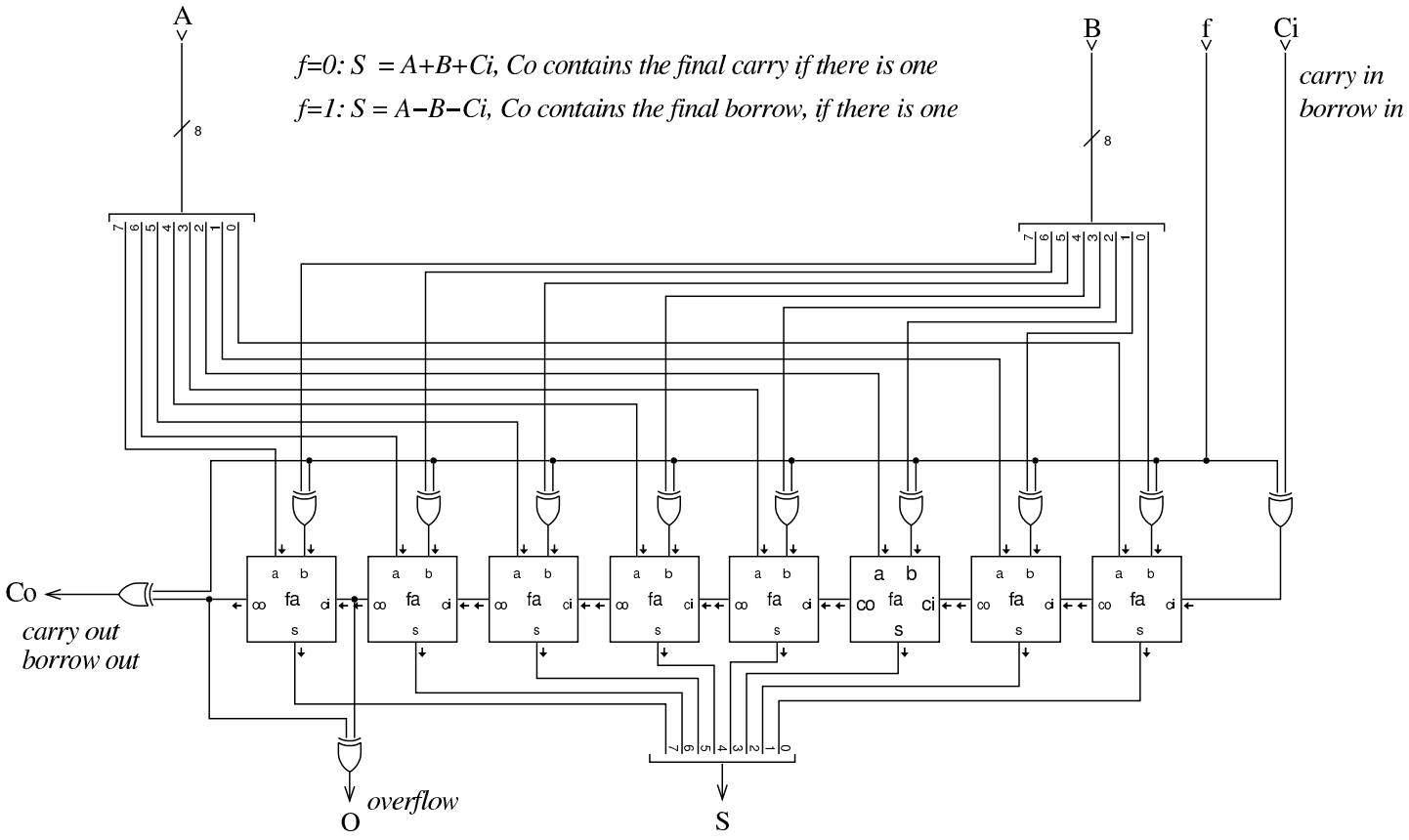
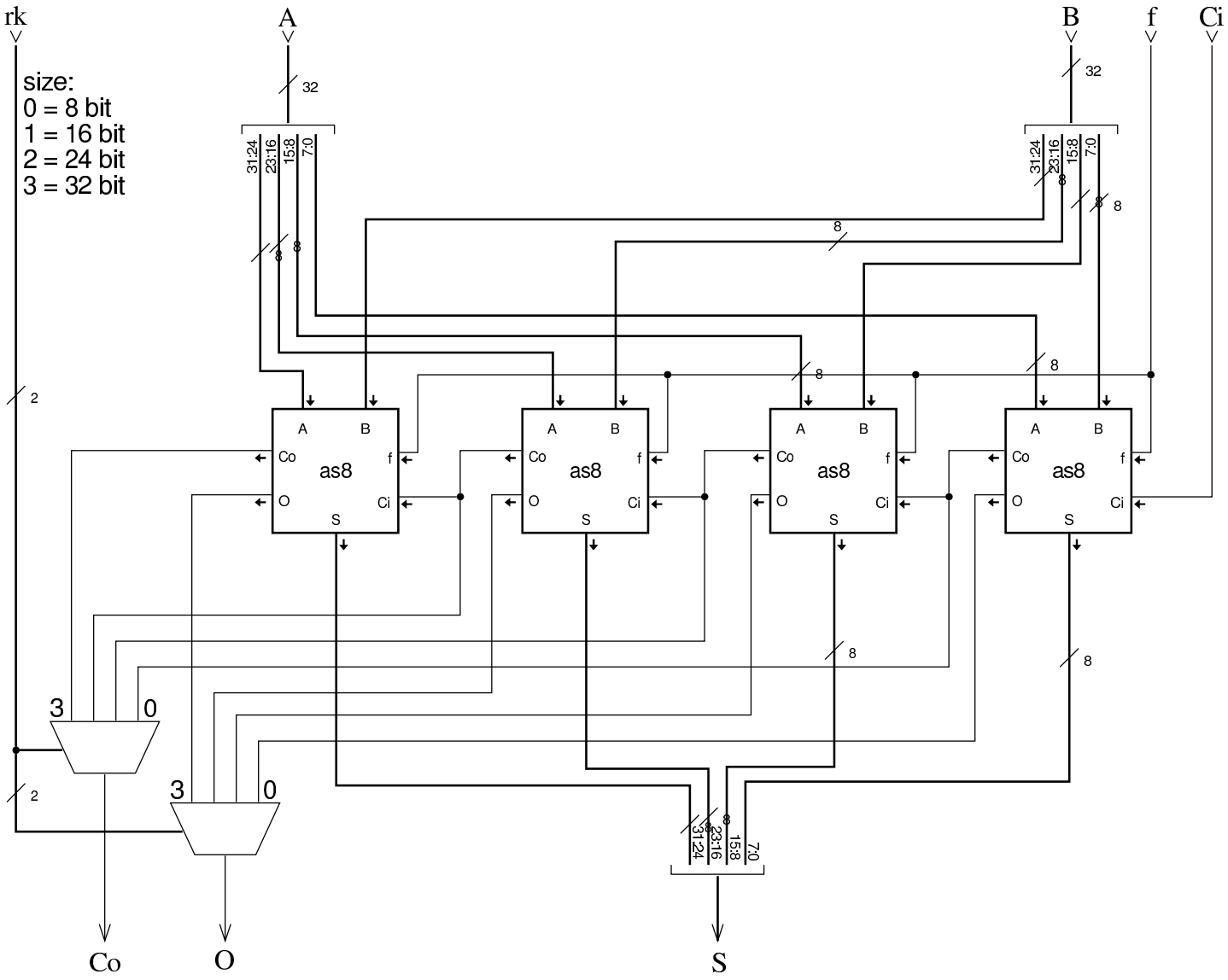


Figure 4.10. Module **as32**: addition and subtraction with the possibility to control the bit size. The module **as8** is described in figure 4.9.



4.5 Shift and rotation

The following figure shows a module that is able to do a logical an arithmetical shift on 8 bits. The ports ***lin***, ***lout***, ***rin*** and ***rout*** (left/right in/out) are there to allow a shift and rotation on multiples of 8 bits. The output ports ***Cl*** and ***Cr*** (carry left/right) contain always the bit value that is shifted outside (on the left or on the right side), whereas the output port ***O*** (overflow) is asserted only when an overflow is produced, due to an arithmetic left shift that changes the sign.

Figure 4.11. Module **sh8**: 8-bit logic and arithmetic shift.

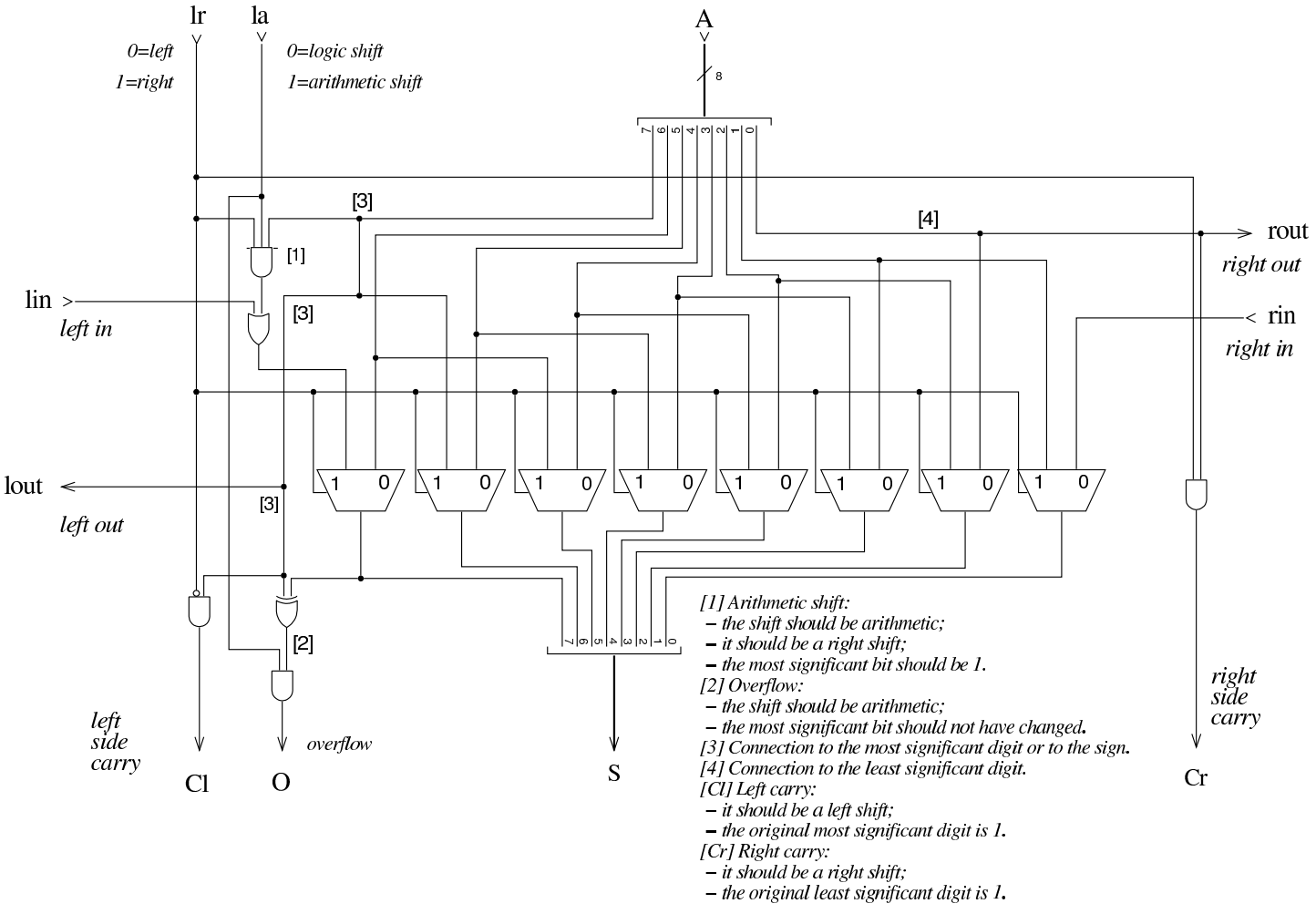


Figure 4.12. Module **sh32**: from 8 to 32-bit logic and arithmetic shift. The ***C*** output (carry) returns the bit value that is shifted outside either on the left or the right side. The module **sh8** is described inside the figure 4.11.

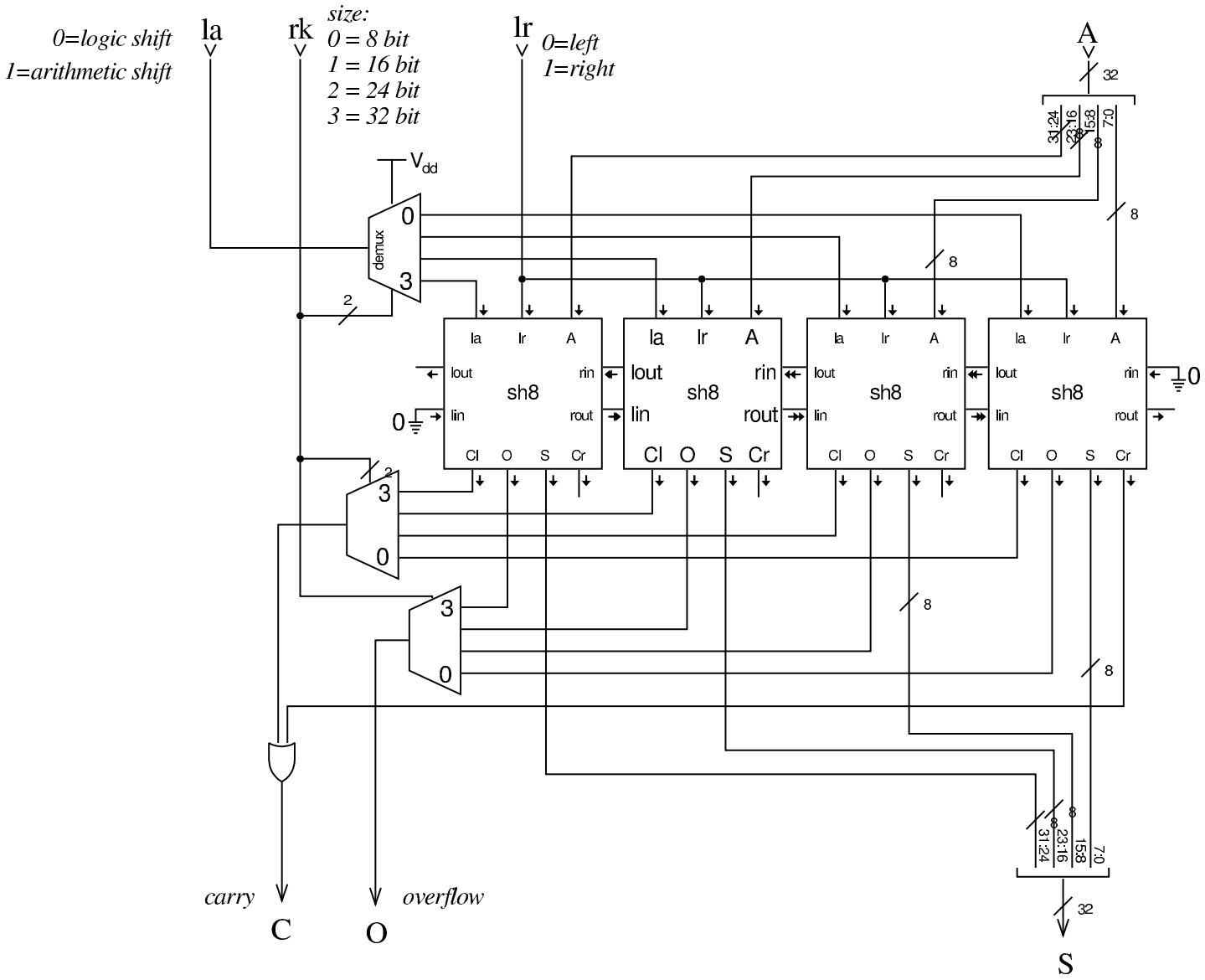


Figure 4.13. Module **`ro32`**: from 8 to 32-bit rotation. As there is no 'arithmetic' rotation, there is no overflow either. The module **`sh8`** is described in figure 4.11.

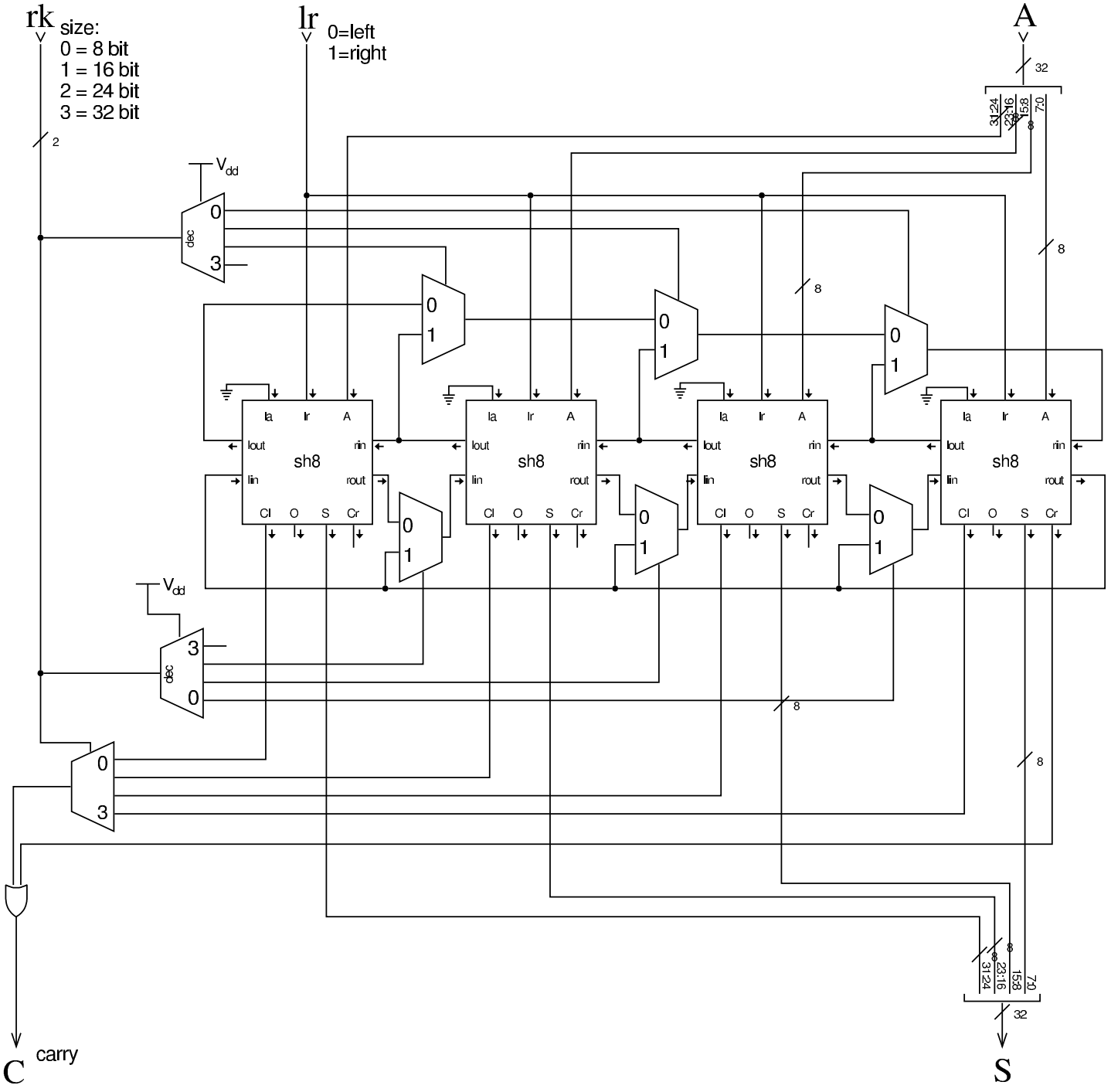
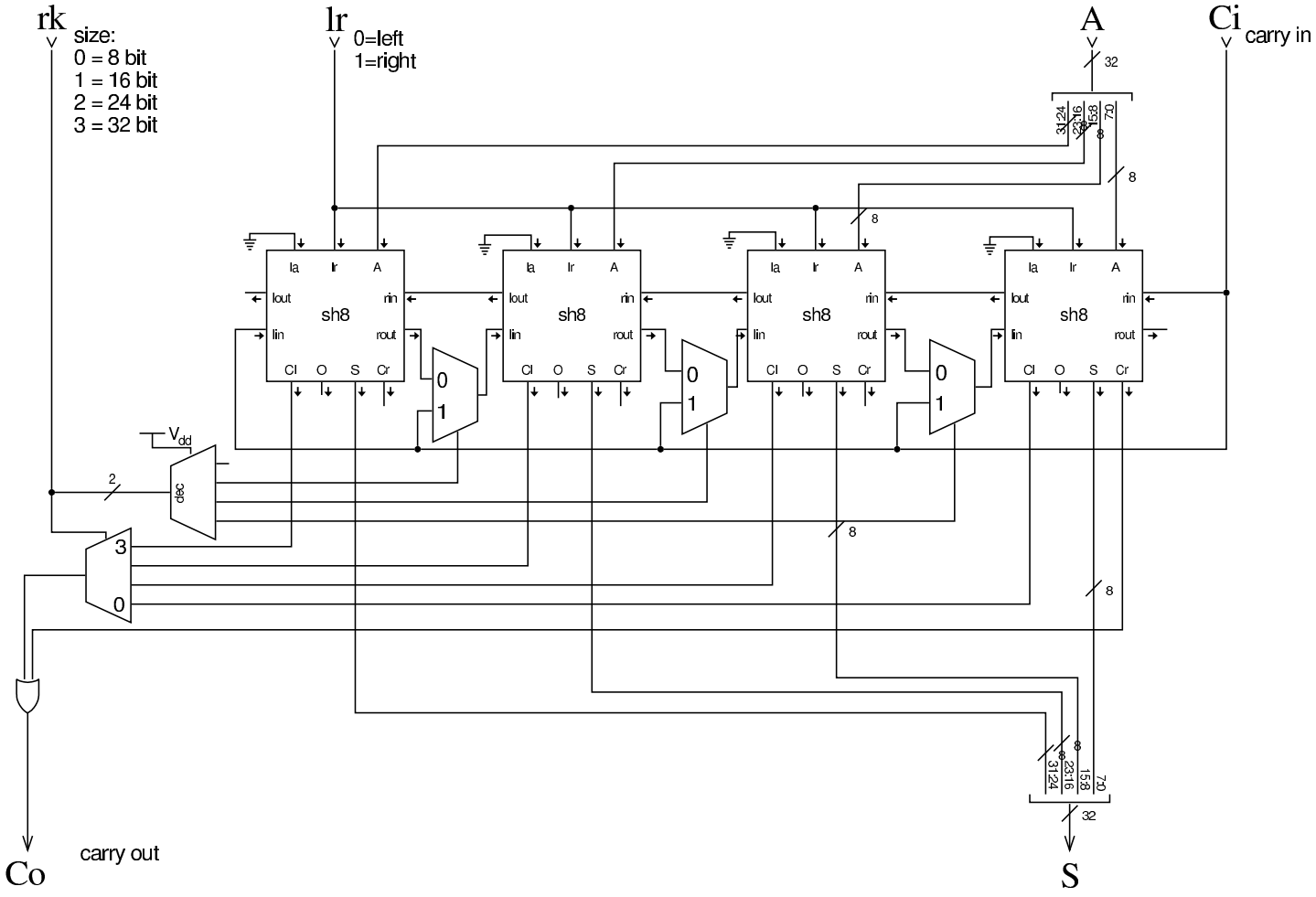


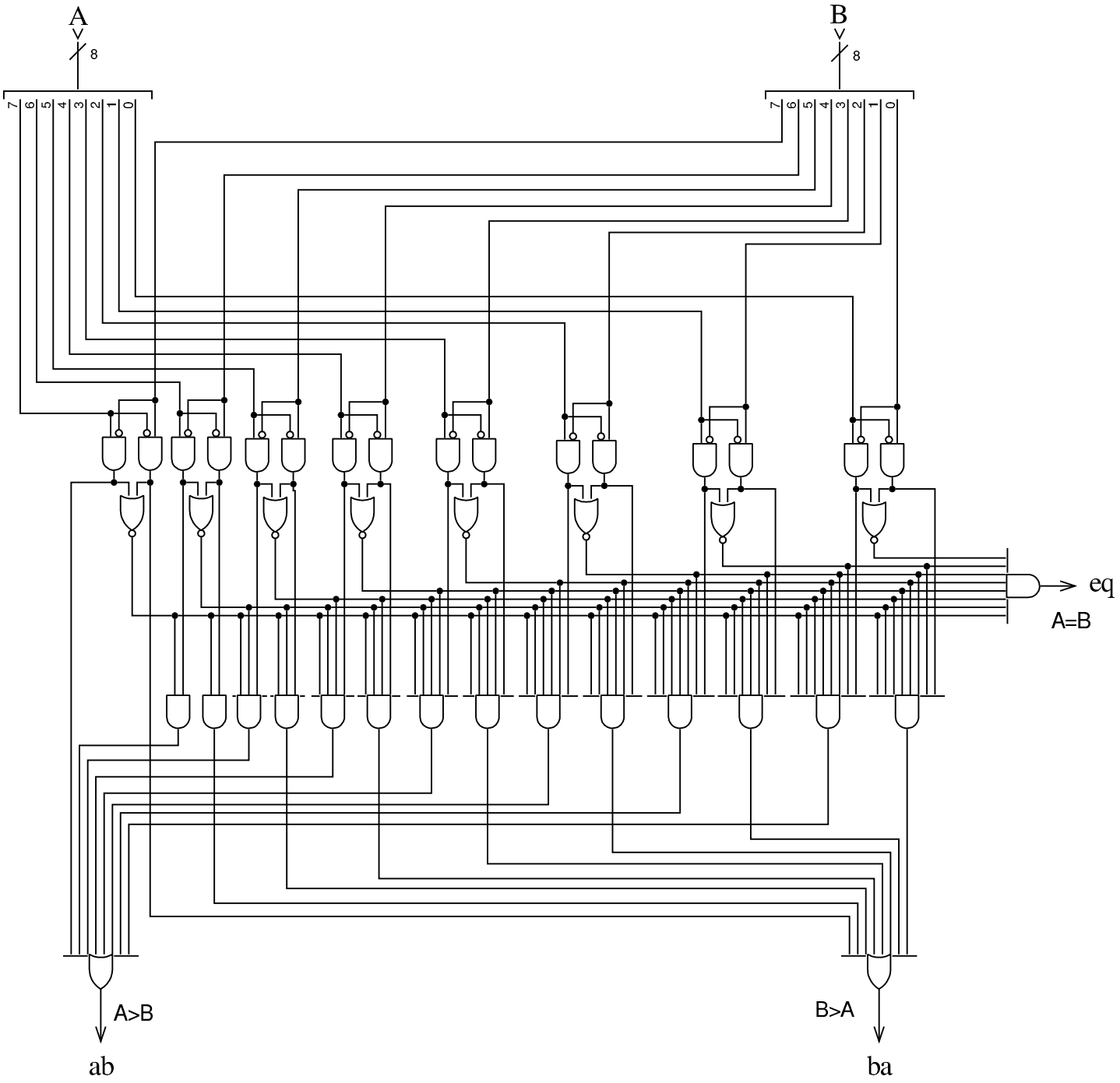
Figure 4.14. Module **rc32**: rotation with carry (the input carry is inserted from one side and the output carry, at the opposite side, is saved for the next rotation). The module **sh8** is described in figure 4.11.



4.6 Magnitude comparation

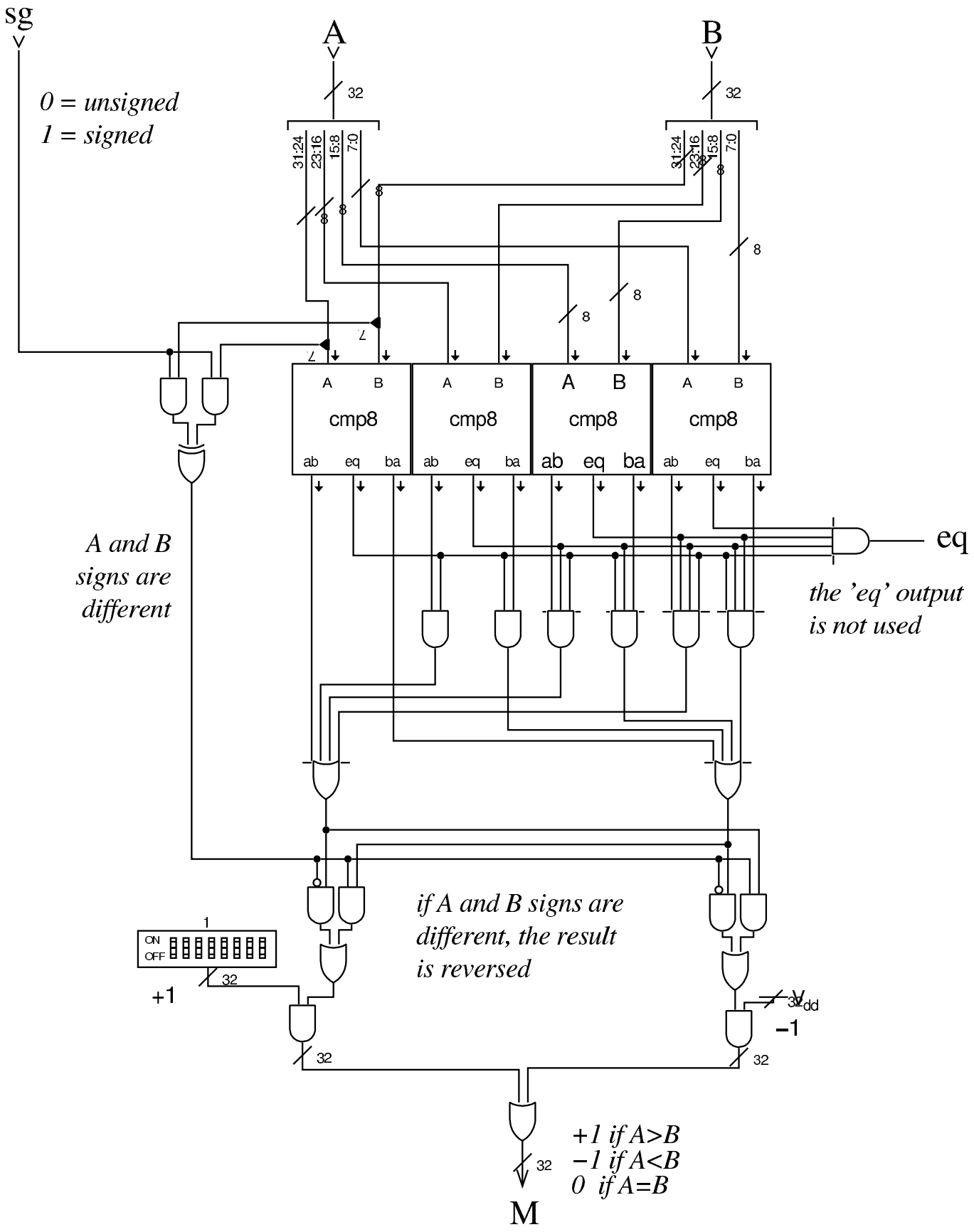
The module **cmp8** compares the magnitude of two unsigned values, asserting a different output port depending on the three possible conditions: $A>B$, $A<B$, $A=B$.

Figure 4.15. Module **cmp8**: 8-bit integer unsigned magnitude comparation.



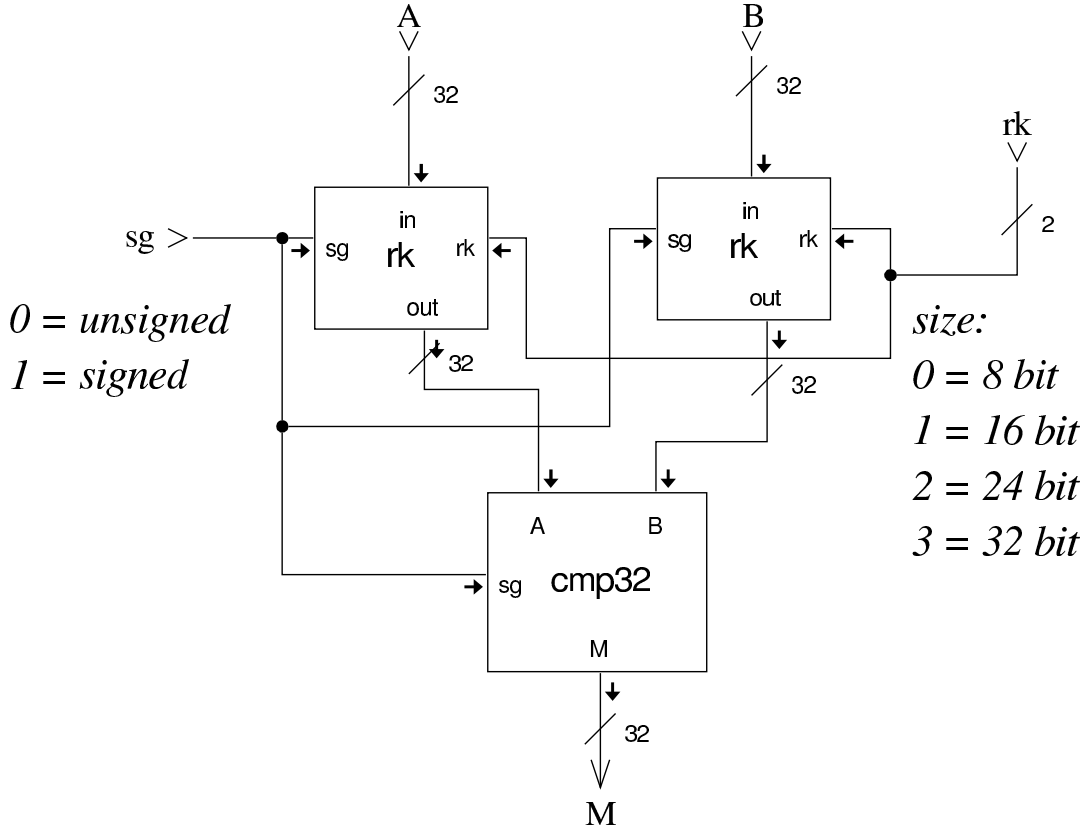
The module **cmp8** can be used to compare two values with a multiple bit size; in that case it is possible to introduce extra logic to consider even the sign: if the input values to be compared are signed and if the signs are different, the result should be reversed. The module **cmp32** compares two 32-bit values, returning a 32-bit value: 0 if the values are equal, +1 if $A>B$; −1 if $A<B$.

Figure 4.16. Module **cmp32**: signed or unsigned 32-bit magnitude compare. The module **cmp8** is described in figure 4.15.



The module **cmp32** does not allow to reduce the bit range to be compared; therefore, if there is the need to control the input bit size, it is necessary to add the module **rk** as a filter (figure 4.5).

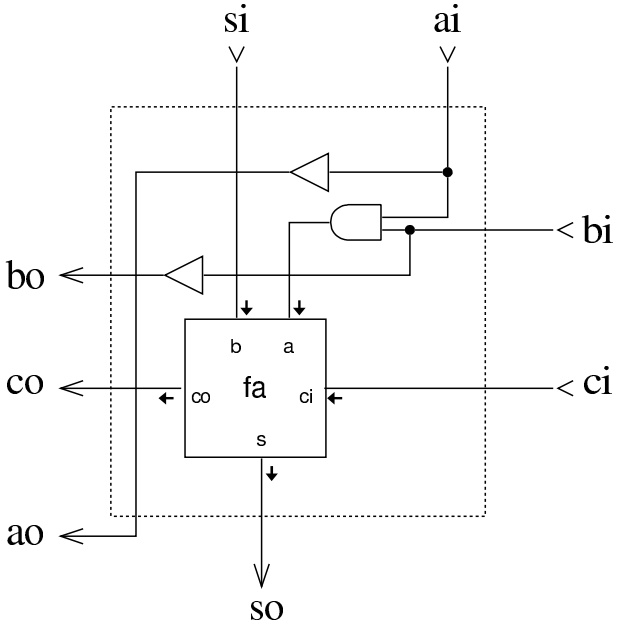
Figure 4.17. Input filter with the module **rk** (figure 4.5).



4.7 Multiplication

As already described in a previous chapter, the multiplication requires a special module that adds the result of the AND operation between the two input values. This module is recalled inside the following figure.

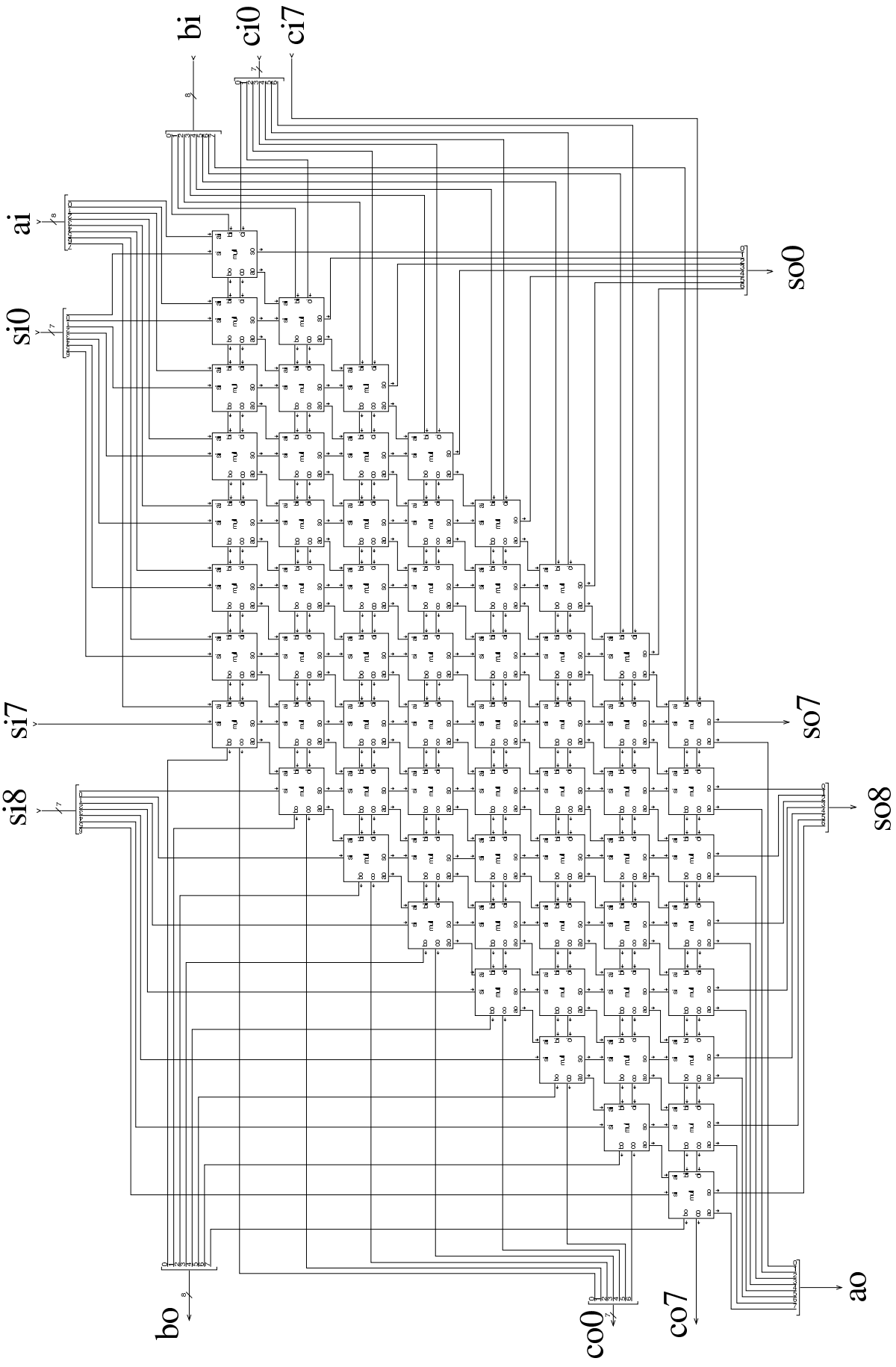
Figure 4.18. Module **`mul`**: 1-bit multiplication. This module is used inside the one that multiplies 8-bit values.



si = sum input
ai = a input = moltiplicand
bi = b input = moltiplicator
ci = carry in
so = sum output = ((ai AND bi) + si + ci)
ao = ai
co = carry out
bo = bi

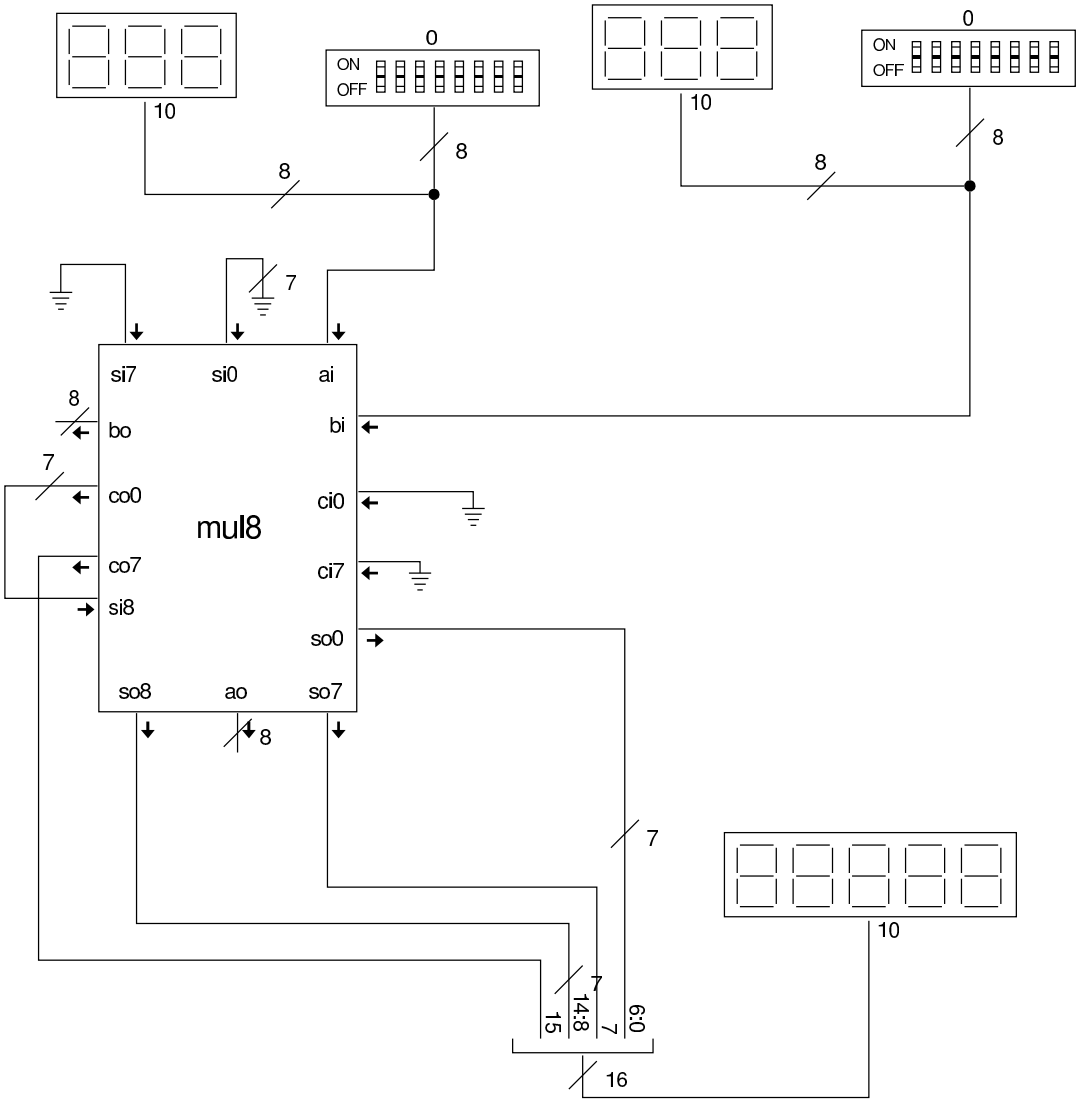
The following figure shows the module **`mul8`**, which is a 8-bit multiplier with all the required ports to connect with other similar modules.

Figure 4.19. Module **`mul8`**: 8-bit multiplication.



As the module **`mul8`** is complex even if used alone, the following figure shows how to use it to multiply 8-bit values.

Figure 4.20. How to use the **`mul8`** alone.



To proceed gradually, the following figure shows a module that multiply 16-bit values, using four **`mul8`** modules.

Figure 4.21. Module **`mul16`**: 16-bit modules. The module **`mul8`** is described in figure 4.19.

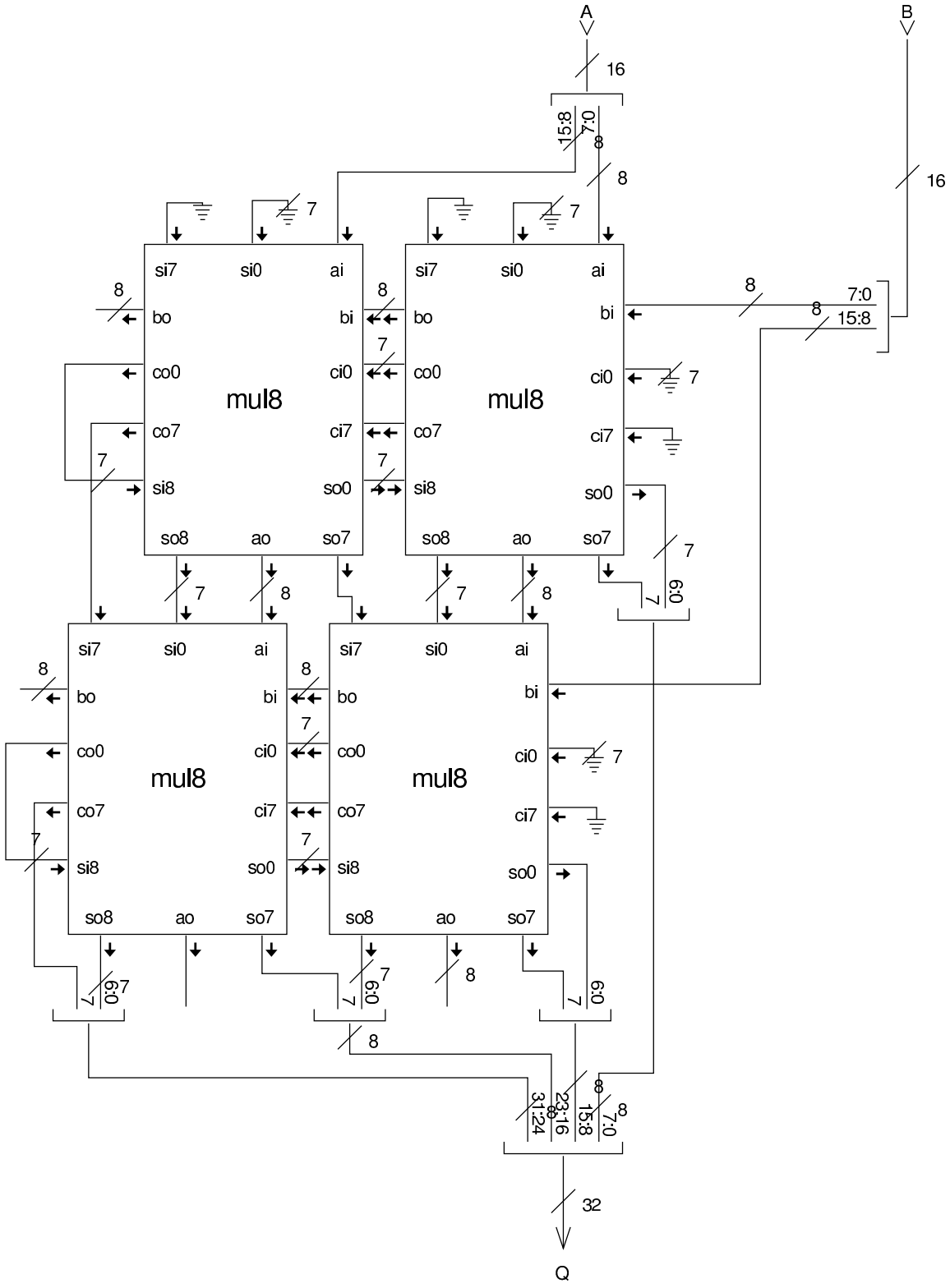
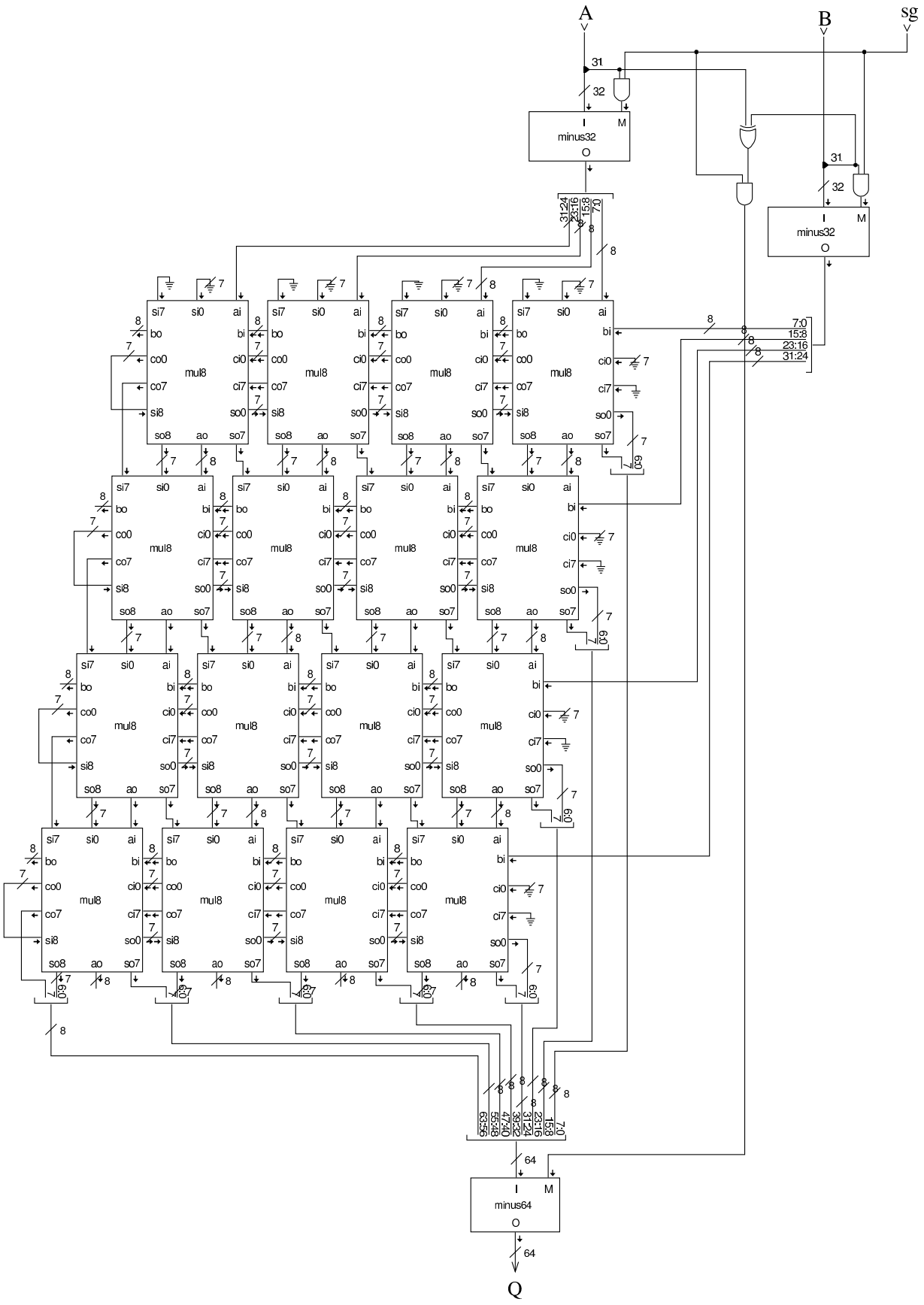


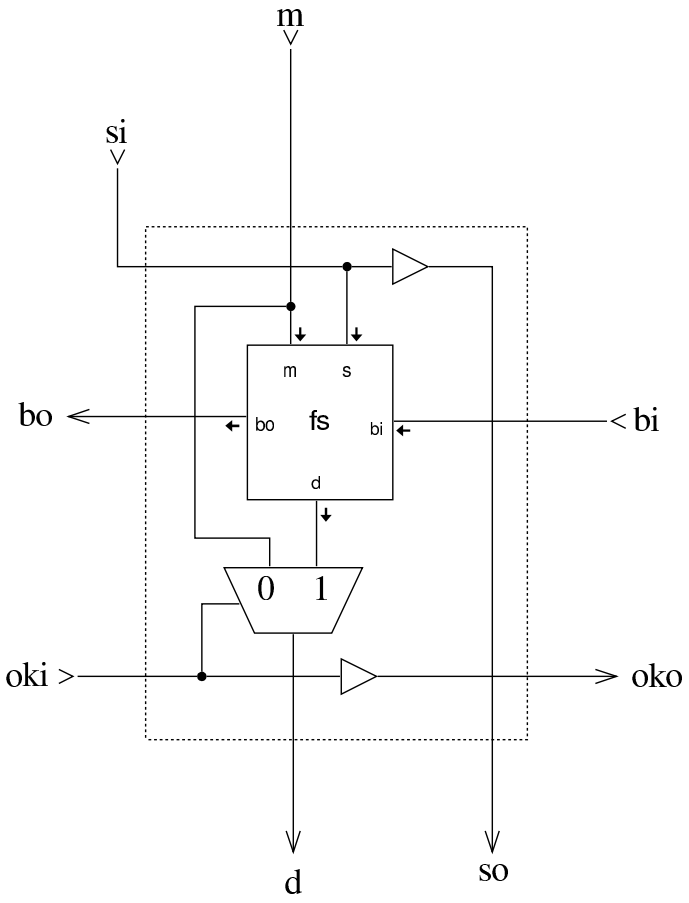
Figure 4.22. Module **`mul32`**: 32-bit signed and unsigned multiplication. The module **`mul8`** is described in figure 4.19, whereas the modules **`minus`**... are described starting from the figure 4.7.



4.8 Division

As already described in a previous chapter, the division requires a special module to subtracts the divisor from the dividend. This module is recalled inside the following figure.

Figure 4.23. Module **`div`**: 1-bit division. This module uses an **`fs`** module, which is a full-subtractor (see section 3.10).



fs = full subtractor
m = minuend
si = input subtrahend
so = output subtrahend
bi = input borrow
bo = output borrow
oki = input ok: the subtraction is valid
oko = output ok: the subtraction is valid

Figure 4.24. Module **`div8`**: 8-bit integer unsigned division. There are 64 **`div`** modules, used to subtract the divisor from the dividend, until the quotient and the remainder are found. The module **`div`** is described in figure 4.23.

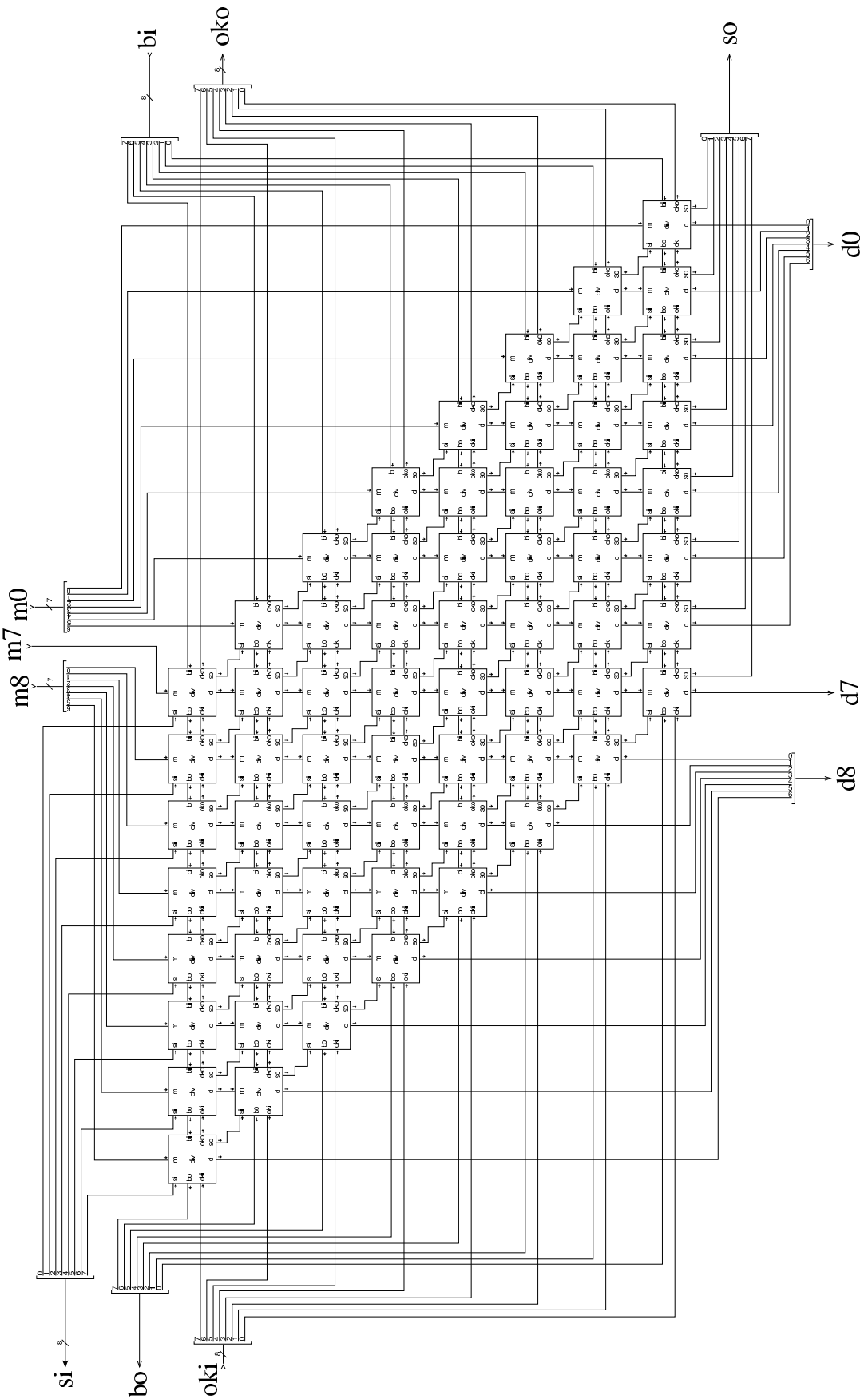


Figure 4.25. Usage example for the module **`div8`** alone. The module **`div8`** is described in figure 4.24.

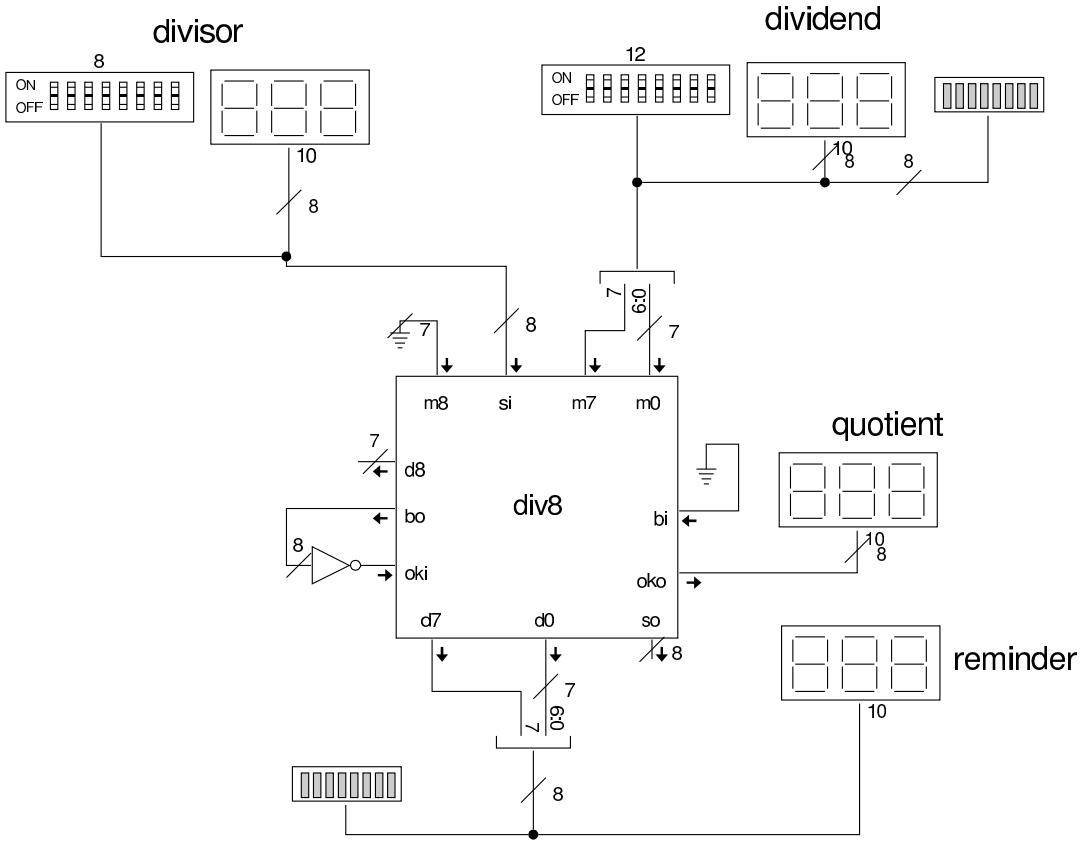


Figure 4.26. Module **`div16`**: 16-bit integer unsigned division. Four modules **`div8`** are used. The module **`div8`** is described inside the figure 4.24.

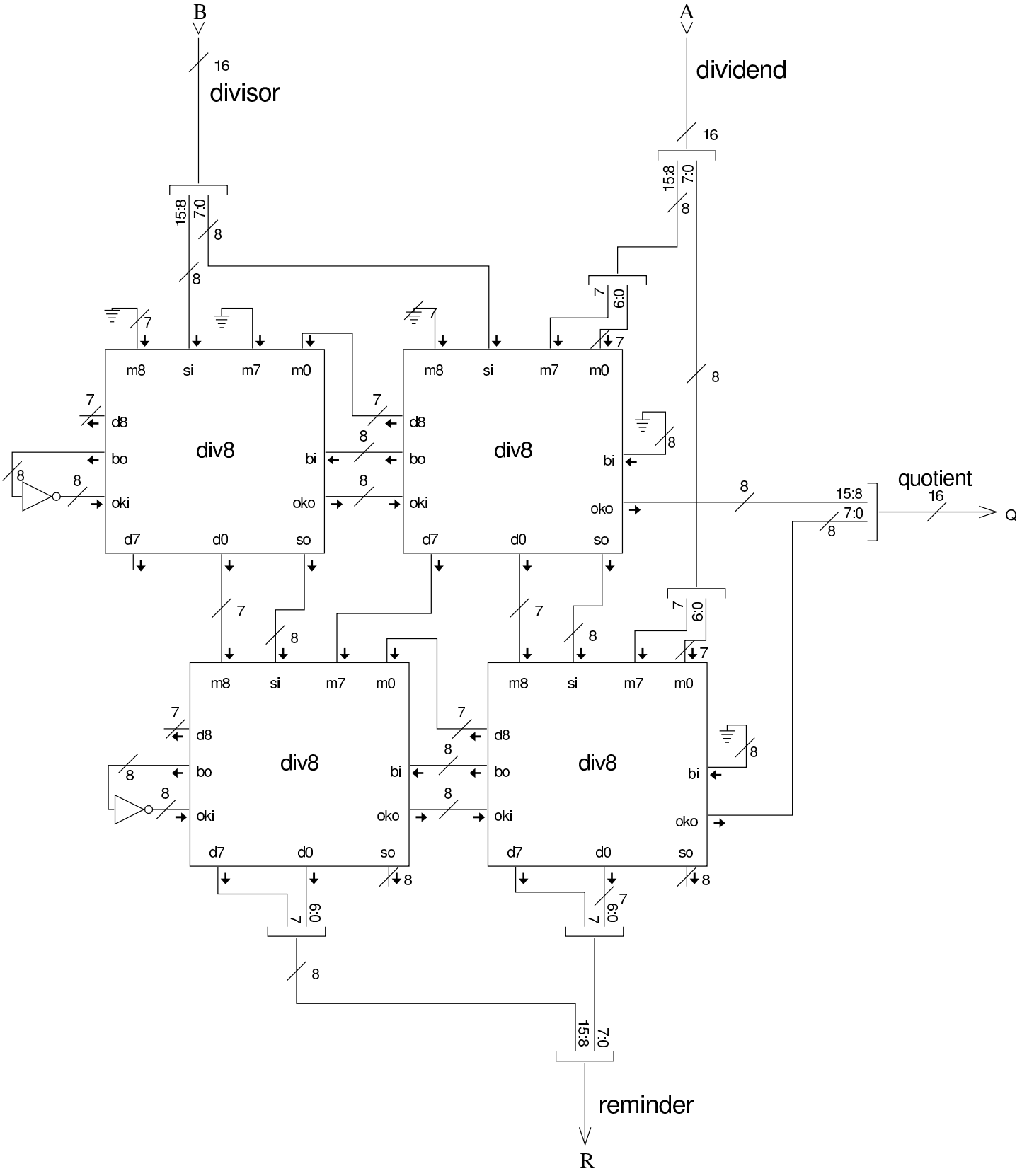
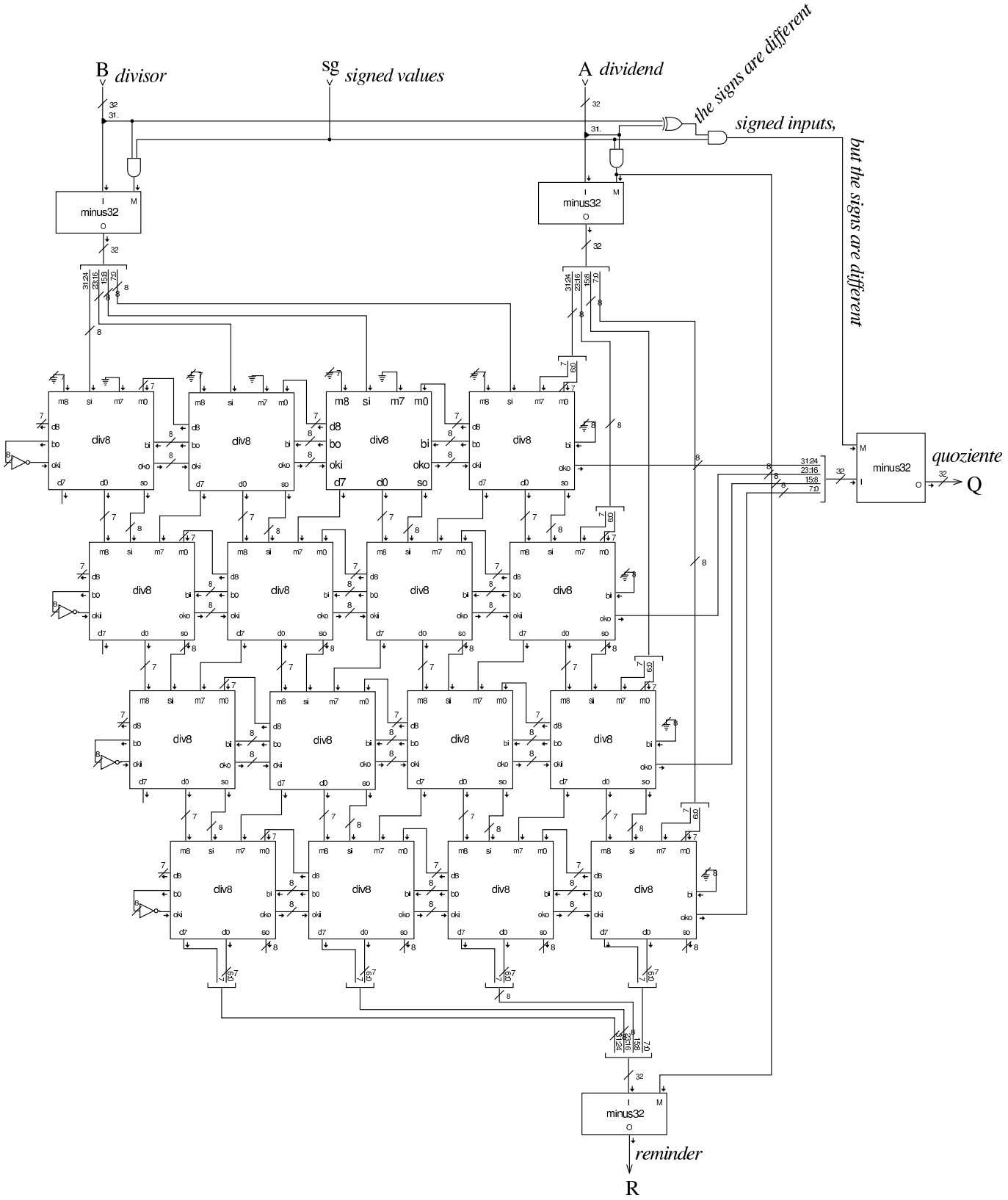


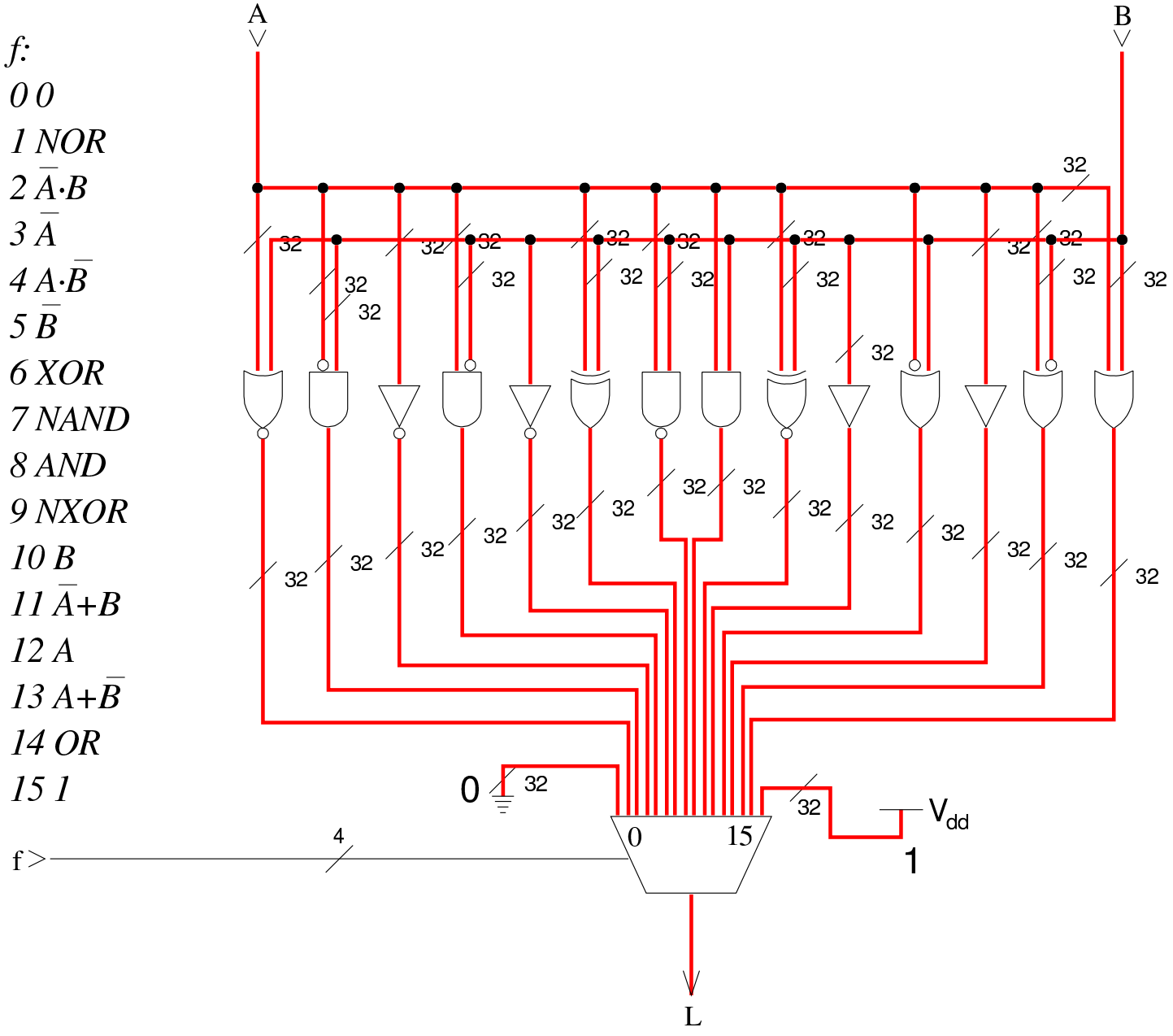
Figure 4.27. Module **`div32`**: 32-bit signed integer division. There are sixteen **`div8`** modules and four **`minus32`** modules to control the sign if the input data is meant to be signed. The module **`div8`** is described in figure 4.24, whereas the module **`minus32`** can be found in figure 4.8.



4.9 Logic unit

A logic unit is a module that calculates bitwise boolean operations. The common operations available are usually NOT, AND, OR and XOR, but the example in the following figure shows all the cases of the table 3.4, even if many of them are unnecessary.

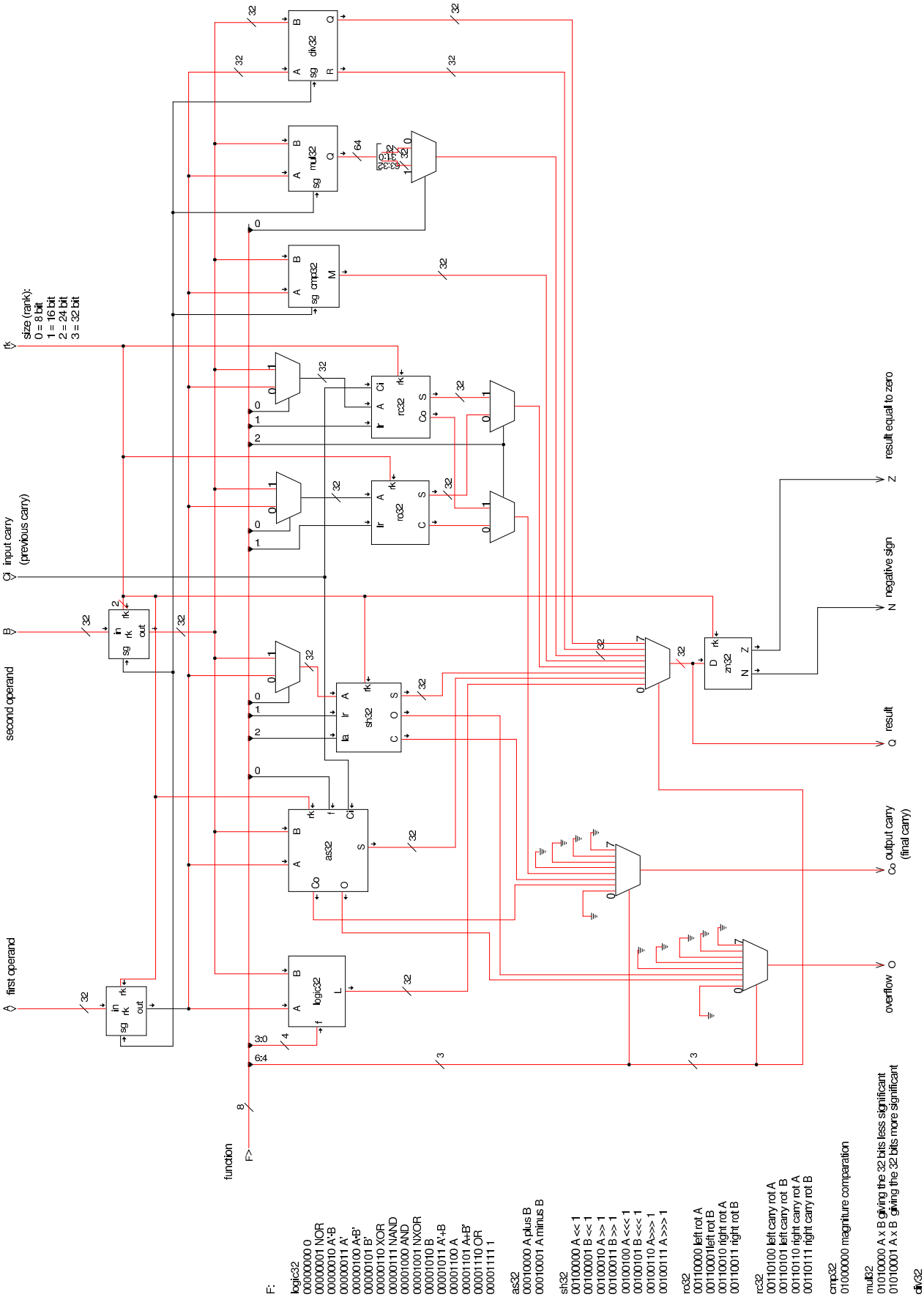
Figure 4.28. Module **`logic32`**: all the sixteen logic operations that can be defined in a combinational circuit with two input ports ad a single output. In this case, every logic gate is multiplied 32 times in paralel.



4.10 Joining all the module together

In the following figure there is an ALU with all the modules described in the previous sections. Please notice that the proposed solution is not optimal and it has only a demonstration purpose.

Figure 4.29. Module **alu32**: 32-bit ALU. The input ***F*** represent the requested function that the ALU should execute.



4.11 Bit sliced ALU

If the requested functions are limited and the addition can be obtained with the simple ripple-carry method, it is possible do design a bit-sliced ALU that is a single bit ALU, which can be expanded to any bit-size.

Figure 4.30. Module **as**: f=0 requires the addition with carry; f=1 requires the subtraction with borrow; if the output carry is different from the input one, there is an overflow.

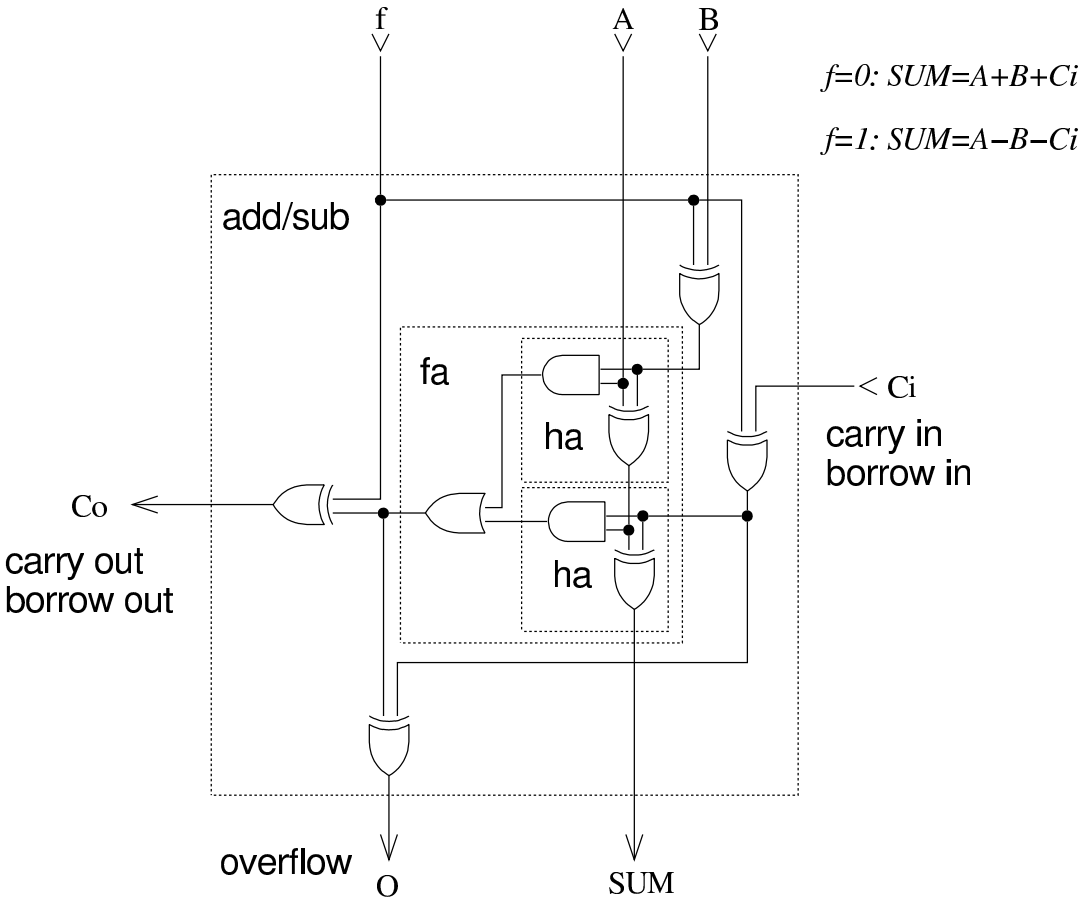


Figure 4.31. Module **`logic`**: 1-bit logic unit. The value received by the input ***f*** selects the logic operation.

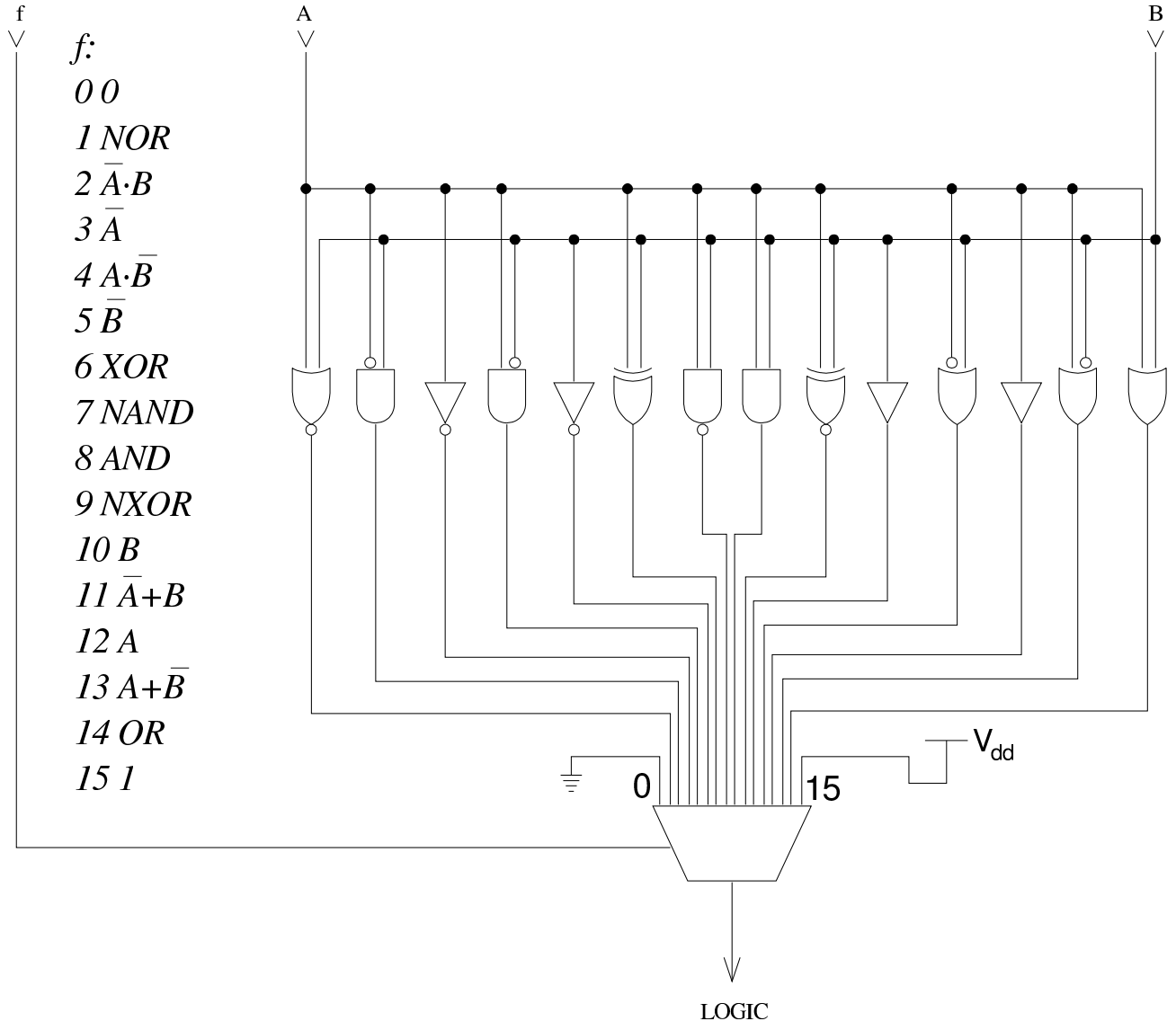


Figure 4.32. Module **`ltgt`**: magnitude comparation. The comparation is made at the bit level, using the input carry if the values are the same. The result is transferred to the next module through the output carry. The module is made of two parts: **`lt`** (A<B) and **`gt`** (A>B). The overall module does the first and the second comparation, based on the function ***f***. The equality match is not done, because it can be obtained with the NXOR operator.

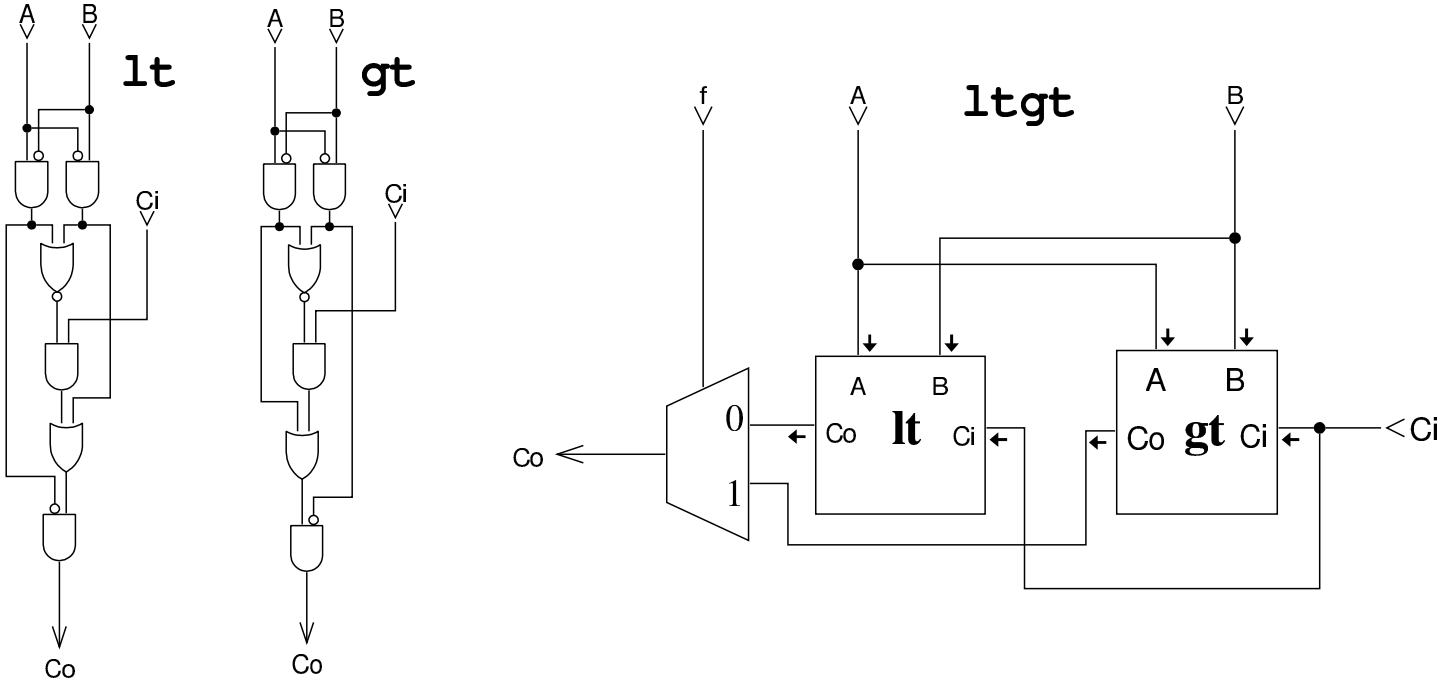


Figure 4.33. Modules **`shl`** and **`shr`**: left and right shift. The shift is done using the carry: the left shift uses the 'normal' carry, whereas the right shift uses a reversed carry line where the names ***Di*** and ***Do*** are used. The shift does not consider the sign; the input ***f*** is used to select the operand to shift: either ***A*** or ***B***.

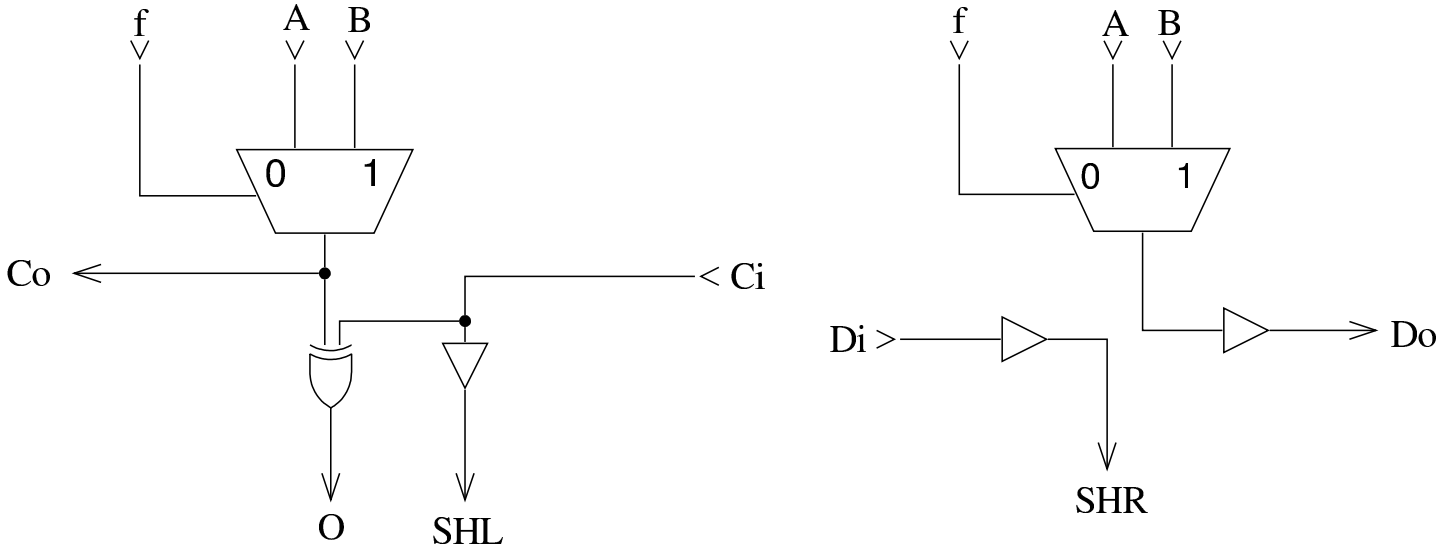


Figure 4.34. Module **z**: zero. It verifies if the value is zero and if all the previous ones are reporting the same (through the carry line). The input *f* allows to select which operand to check.

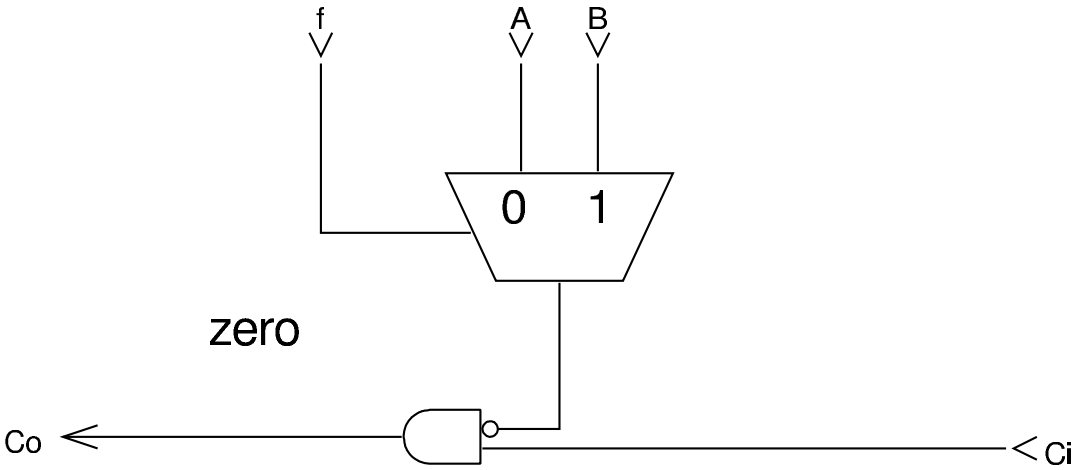


Figure 4.35. Module **alu**: 1-bit ALU, made of the previous modules.

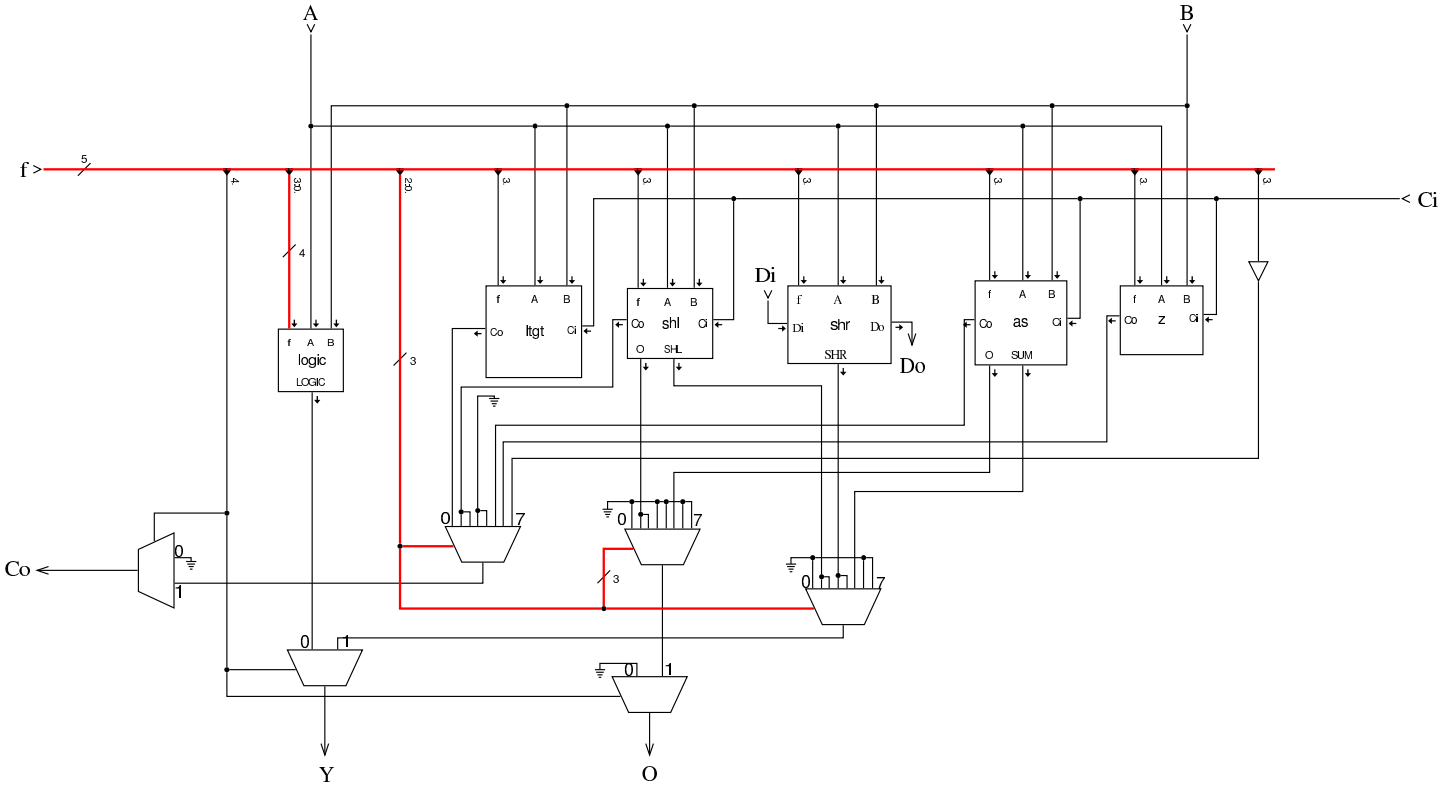
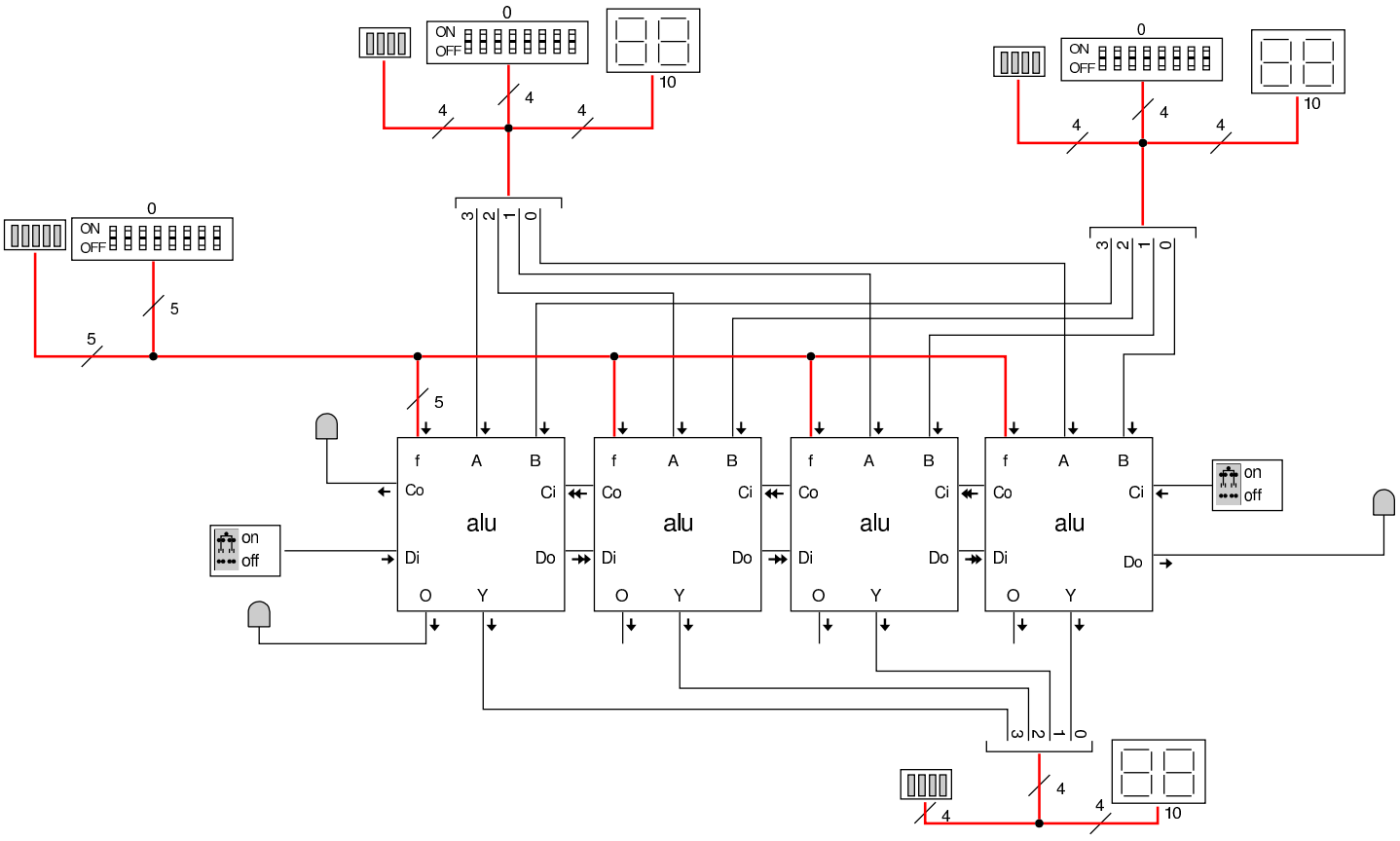


Figure 4.36. 4-bit ALU using four **`alu`** modules.



Chapter 5
Latch and flip-flop

5.1 Propagation delay 185
5.2 Truth tables 187
5.3 SR latch 188
5.4 Bounce-free switch 193
5.5 Gated SR latches 193
5.6 SR flip-flop 195
5.7 Time: setup/hold and recovery/removal 198
5.8 D latch and D flip-flop 199
5.9 T flip-flop 203
5.10 JK flip-flop 204
5.11 Troublesome JK flip-flops 207

bounce free 193 delay 185 D flip-flop 199 Earle latch 199 edge-triggered 195 flip-flop 185 195 gated D latch 199 hold time 198 JK flip-flop 204 207 latch 185 master-slave 199 memory 185 propagation delay 185 recovery time 198 removal time 198 sequential circuit 185 setup time 198 SR flip-flop 195 SR latch 188 switch 193 transparent latch 199 T flip-flop 203

The combinational circuits transform the input data into the output data, according to a certain function, without taking into account any previous status: the data is translated always in the same way, after a little amount of time. Because of this characteristic, the combinational circuits are said to be memoryless. Instead, a ***sequential circuit*** takes into account the dynamic on which the input data is received, keeping a ***status***, which is updated by the circuit itself and which is in fact an additional source of input data that influences the output result.

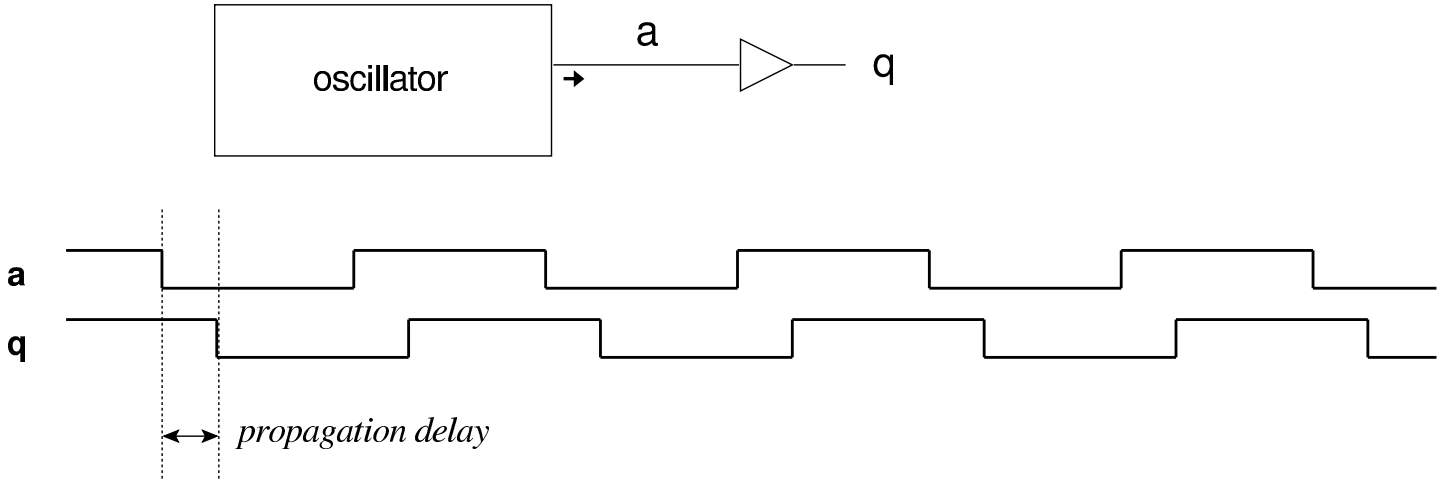
The sequential circuits are based on components known as ***memories***, which are made of ***latches*** or ***flip-flops***. A latch is a circuit whith some kind of feedback, by which one or more output lines becomes input again; a flip-flop is a sophisticated latch which is responsive to the variation of a data.

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

5.1 Propagation delay

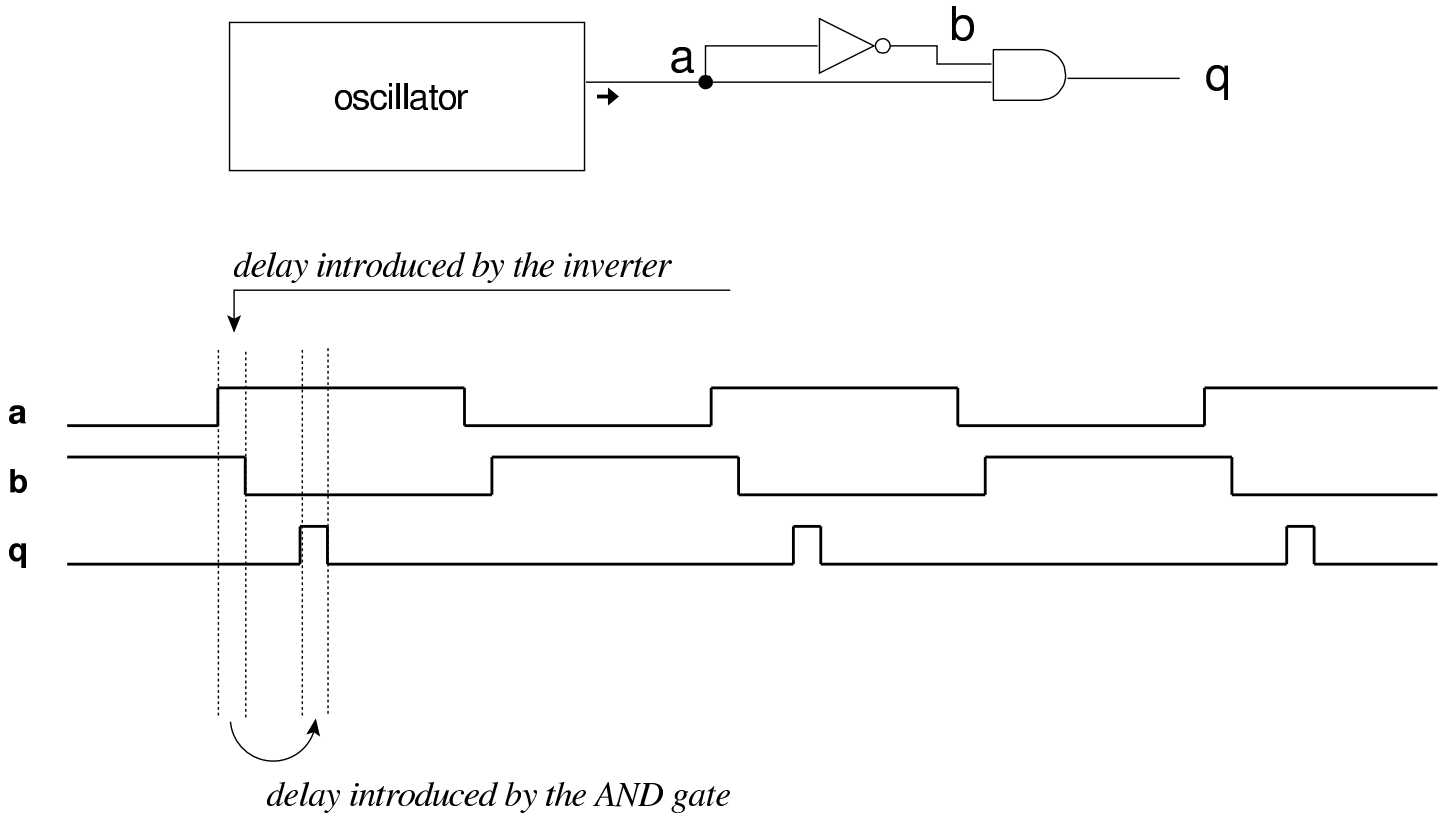
Every combinational circuit reacts to the input data change updating the output with a little time delay, known as the ***propagation delay***. The amount of the delay time depends on the physical and mechanical characteristics of the real circuit.

Figure 5.1. An impulse signal traversing a buffer shows on the output a little delay.



The propagation delay might be used to produce a short impulse, as it shows the following figure.

Figure 5.2. A short impulse produced by the delay propagation.



5.2 Truth tables

The truth table is a way to explain the combinational circuit behaviour, whereby the output values are shown as functions of the input ones, without considering either the propagation delay or the dynamic that the input data might have. To explain a sequential circuit behaviour a kind of truth table might be used, but if they are used, the notations should be understood depending on the particular context. The following sections introduce latches and flip-flops, which might be influenced by the input data signal variations, therefore the following notation is used:

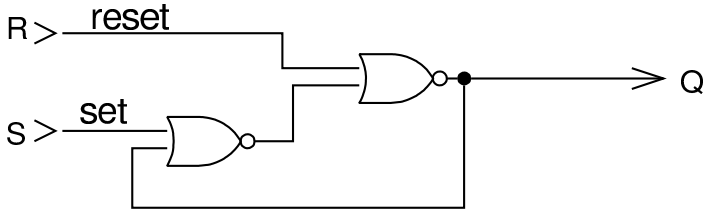
0	stable 0
_/‾	variation from 0 to 1, or positive edge
1	stable 1
‾_	variation from 1 to zero, or negative edge

5.3 SR latch

«

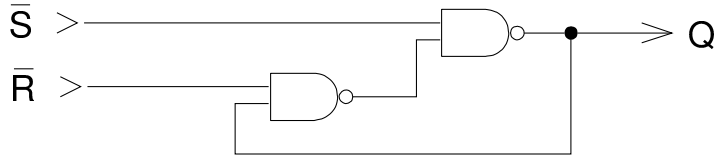
The simplest sequential circuit that keeps its status memory is the one shown on the following two figures, made either with NOR or NAND gates. With the NOR gates example, the ***Q*** output is initially unknown and might be either 0 or 1; then, asserting the ***S*** input (set) even with a short positive impulse, the ***Q*** output is asserted (activated) and it remains active until the ***R*** input (reset) remains negated. That is: a short positive impulse to the ***S*** input sets the output, whereas the same impulse to the ***R*** input resets the output. With the example with NAND gates, the inputs $\overline{S}$ and $\overline{R}$ are negated, so that the set and reset impulse should be negative.

Figure 5.5. Simple set-reset latch with NOR gates.



S	R	Q
0	0	*unchanged*
0	_/‾	0
0	1	0
0	‾_	0
_/‾	0	1
_/‾	_/‾	0
_/‾	1	0
_/‾	‾_	1
1	0	1
1	_/‾	0
1	1	0
1	‾_	1
‾_	0	1
‾_	_/‾	0
‾_	1	0
‾_	‾_	*not allowed!*

Figure 5.6. Simple set-reset circuit with NAND gates.

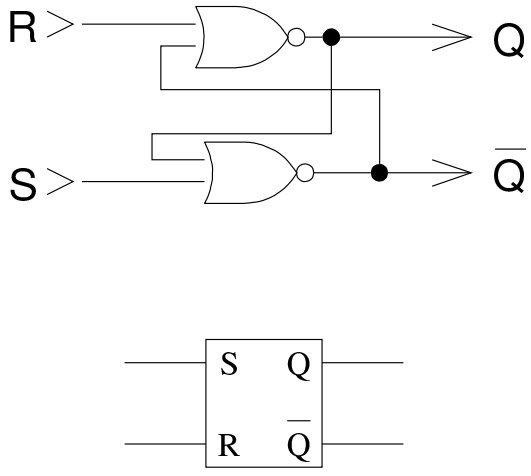


$\overline{S}$	$\overline{R}$	Q
0	0	1
0	_/‾	1
0	1	1
0	‾_	1
_/‾	0	0
_/‾	_/‾	*non allowed!*
_/‾	1	1
_/‾	‾_	0
1	0	0
1	_/‾	0
1	1	*unchanged*
1	‾_	0
‾_	0	1
‾_	_/‾	1
‾_	1	1
‾_	‾_	1

The two figures above show a circuit and the corresponding truth table, which highlights a condition that is not allowed: the NOR circuit should not let both inputs change from 0 to 1 simultaneously, whereas the NAND one should not let both input change from 0 to 1 simultaneously. The reason for the not allowed condition is explained later.

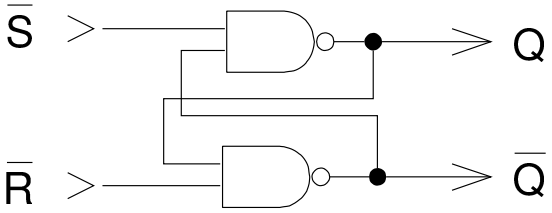
The above figures represent a simplified version of kind of circuit known as ***SR latch***, which usually has another negated output, $\overline{Q}$, which gets the opposite value of Q.

Figure 5.7. SR latch with NOR gates.



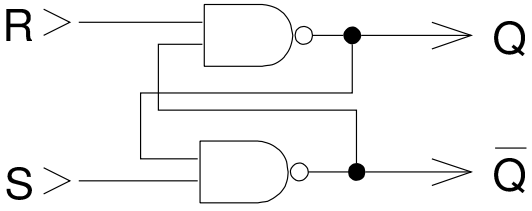
S	R	Q	$\overline{Q}$
0	0	*unchanged*	
0	_/‾	0	1
0	1	0	1
0	‾_	0	1
_/‾	0	1	0
_/‾	_/‾	0	1
_/‾	1	0	1
_/‾	‾_	1	0
1	0	1	0
1	_/‾	0	1
1	1	0	0
1	‾_	1	0
‾_	0	1	0
‾_	_/‾	0	1
‾_	1	0	1
‾_	‾_	*not allowed!*	

Figure 5.8. SR latch made with NAND gates.



$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$
0	0	1	1
0	_/‾	1	0
0	1	1	0
0	‾_	1	0
_/‾	0	0	1
_/‾	_/‾	*not allowed!*	
_/‾	1	1	0
_/‾	‾_	0	1
1	0	0	1
1	_/‾	0	1
1	1	*unchanged*	
1	‾_	0	1
‾_	0	1	0
‾_	_/‾	1	0
‾_	1	1	0
‾_	‾_	1	0

Figure 5.9. SR latch with NAND gates, changing the meaning of the input lines.

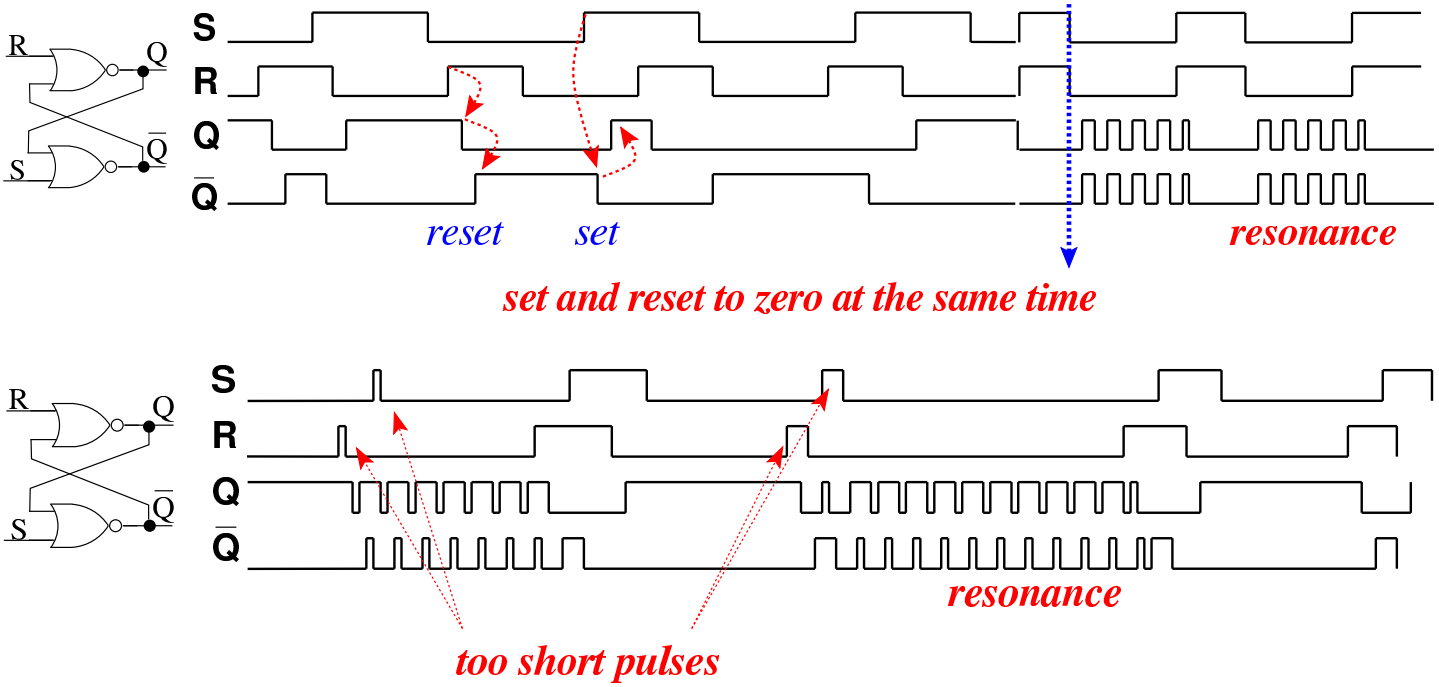


S	R	Q	$\overline{Q}$
0	0	1	1
0	_/‾	0	1
0	1	0	1
0	‾_	0	1
_/‾	0	1	0
_/‾	_/‾	*not allowed!*	
_/‾	1	0	1
_/‾	‾_	1	0
1	0	1	0
1	_/‾	1	0
1	1	*unchanged*	
1	‾_	1	0
‾_	0	1	0
‾_	_/‾	0	1
‾_	1	0	1
‾_	‾_	1	0

The SR latch made with NAND gates is equivalent to the NOR one with the input ports negated; however, the SR latch with NAND gates is also used changing the input port names as it is shown in the last figure above, but this way the behaviour is no more equal to the NOR gate version, although it is nearly the same. For that reason, when an SR latch is drawn as a box, it is necessary to specify which truth table is used; anyway, the SR latch alone is used very seldom and it is advisable to

depict it with logic gates and not only as a box.

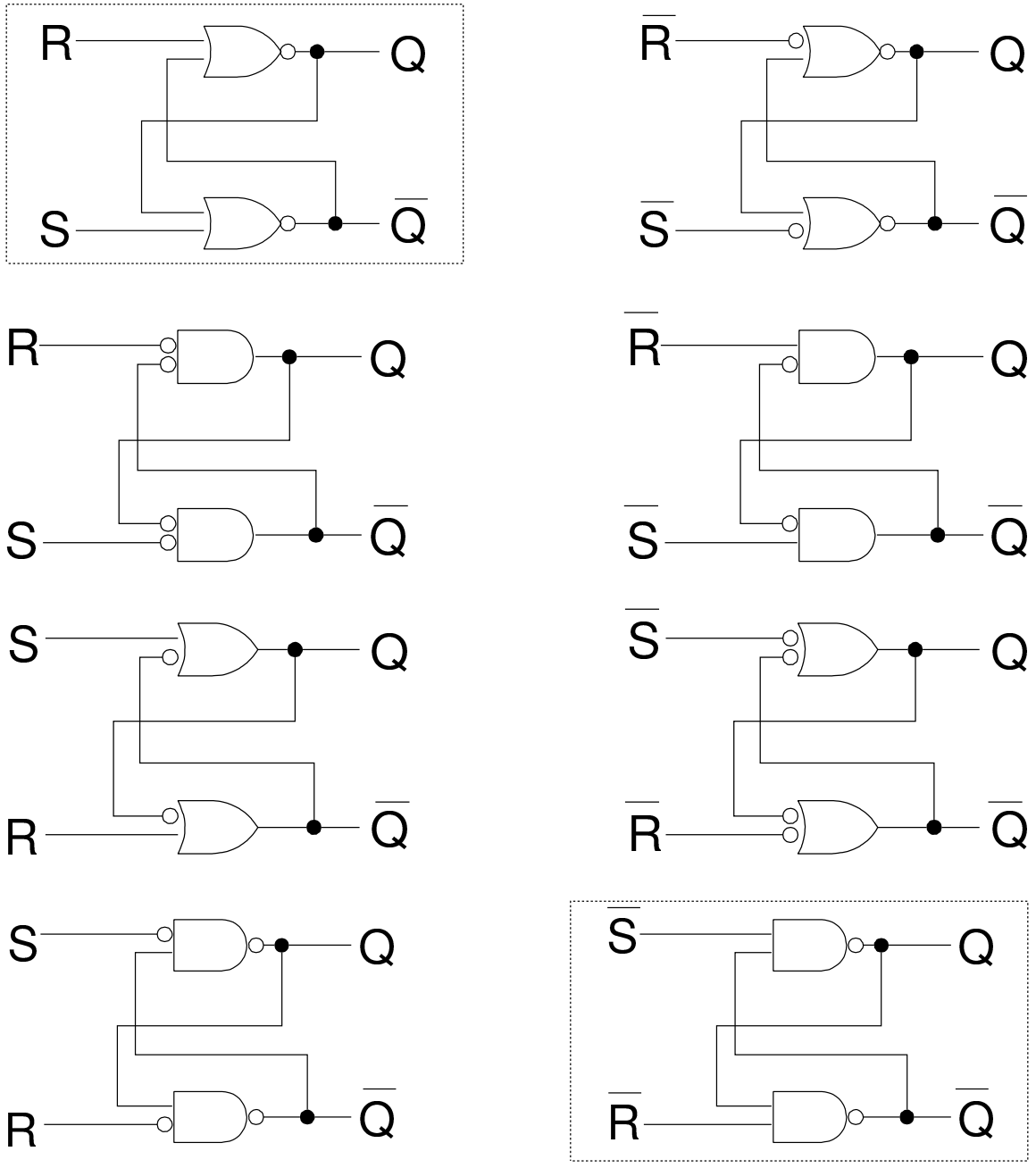
Figure 5.10. NOR SR latch traces showing two critical situations: the first time both the input ports are asserted and becomes negated simultaneously; the second time the inputs impulse is too short.



The figure above shows a NOR SR latch timing diagram, where the propagation delay is evidenced, but especially is visible what happens when the input ports drop to zero simultaneously: because of the propagation delay, the two outputs remains for a while to zero, but then, as the inputs are now zero, the outputs are activated and that start a loop of deactivation and activation. This is the problem that imposes to avoid to drop to zero the input ports simultaneously. But even a too short pulse might create troubles: the input impulse should be long enough to allow the latch to change its state (if that impulse should change it).

The SR latch is the basis for all other latches and flip-flops; that is why it is important to know in how many ways it can be implemented, as the following figure shows.

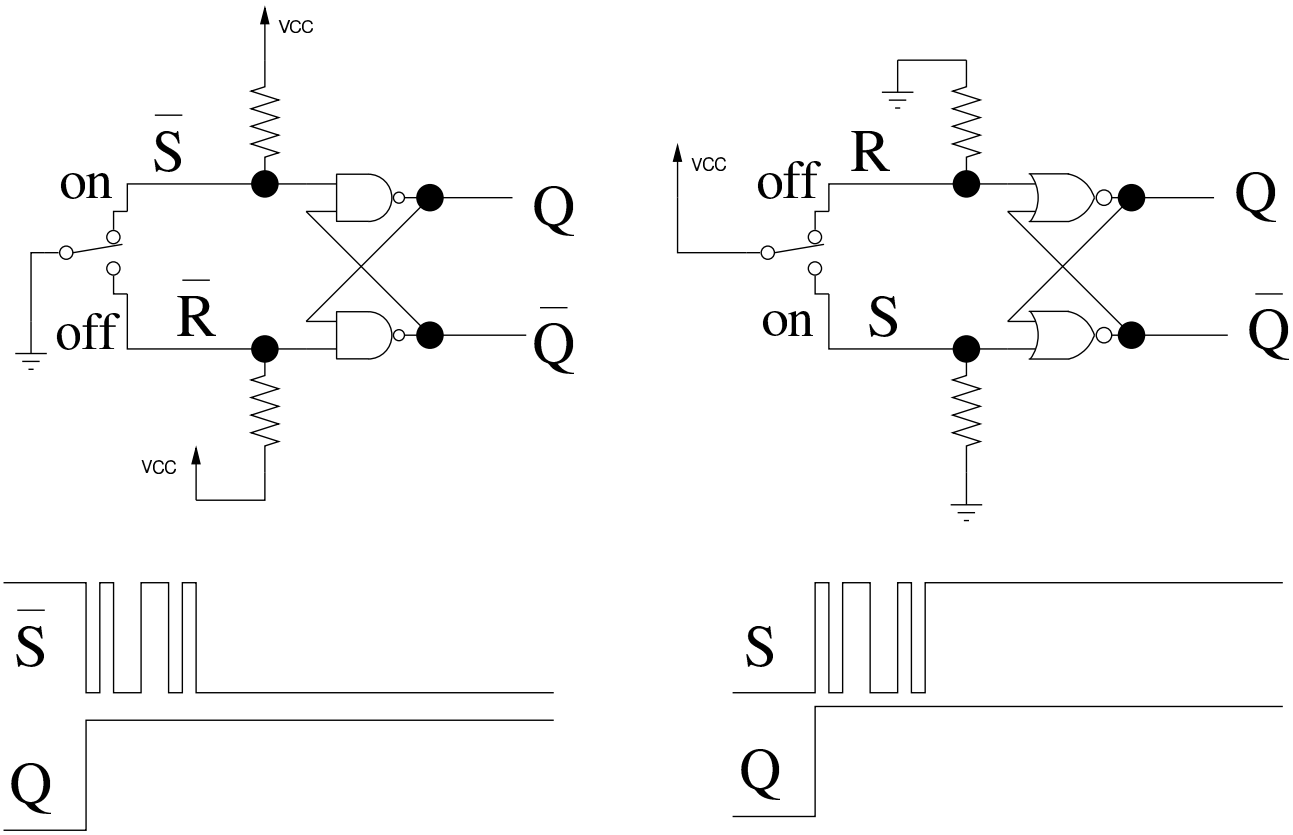
Figure 5.11. SR latch in many equivalent implementations, where care should be given to the input ports order. On the left side there are the active-high input versions, whereas on the right side appear the ones with active-low inputs. The most common SR latches are highlighted: NOR and NAND.



5.4 Bounce-free switch

When building real electronic logic circuits, switches are often used, but in the real world, these components have the ‘bounce’ problem, which means that opening or closing the switch some unwanted impulses are generated. To avoid these impulses the SR latches are used, for example in the way shown by that the figure below.

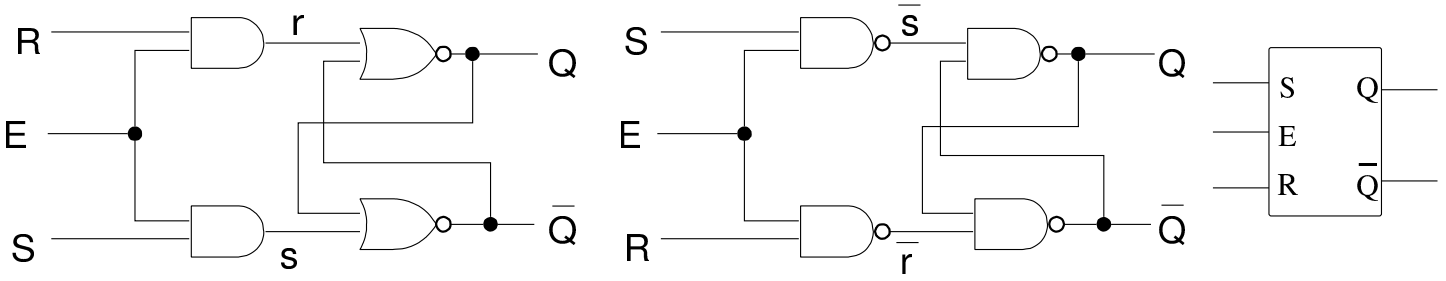
Figure 5.12. SR latch used to filter the bounces produced by a switch.



5.5 Gated SR latches

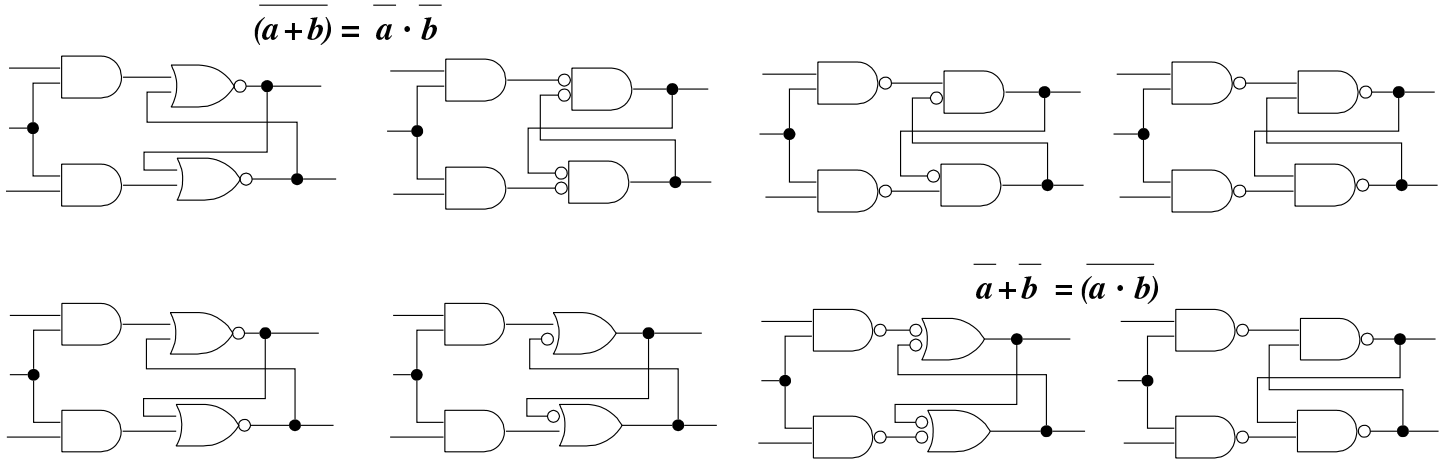
The SR latch can be extended including two control gates to enable the set-reset input lines. In practice the new SR latch has an additional input port used to enable the other two inputs: when the enable input is asserted, the SR latch work as usual, but when it is negated, the other input lines are just ignored.

Figure 5.13. Gated SR latch: when the ***E*** input (enable) is asserted, it works as the usual SR latch; when the ***E*** input is negated, the other inputs are ignored. The circuit appears in the two common versions, together with the usual symbology.



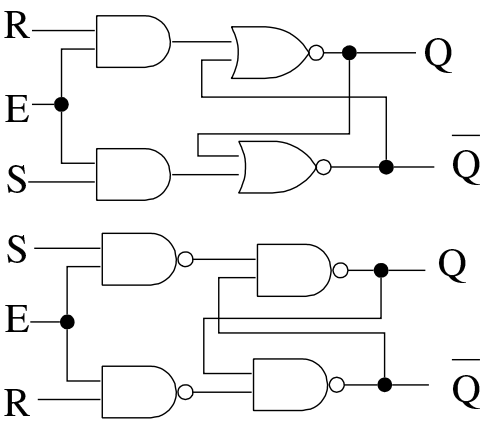
The two diagrams in the above figure are equivalent and it can be demonstrated with the De Morgan's laws.

Figure 5.14. Gated SR latches equivalences.



The gated SR latch does not resolve the resonance triggering problem already described: the enable input should be asserted only when the other inputs are in a valid condition, avoiding the case when both ***S*** and ***R*** are asserted at the same time, because when the enable input becomes negated, it would trigger the resonance.

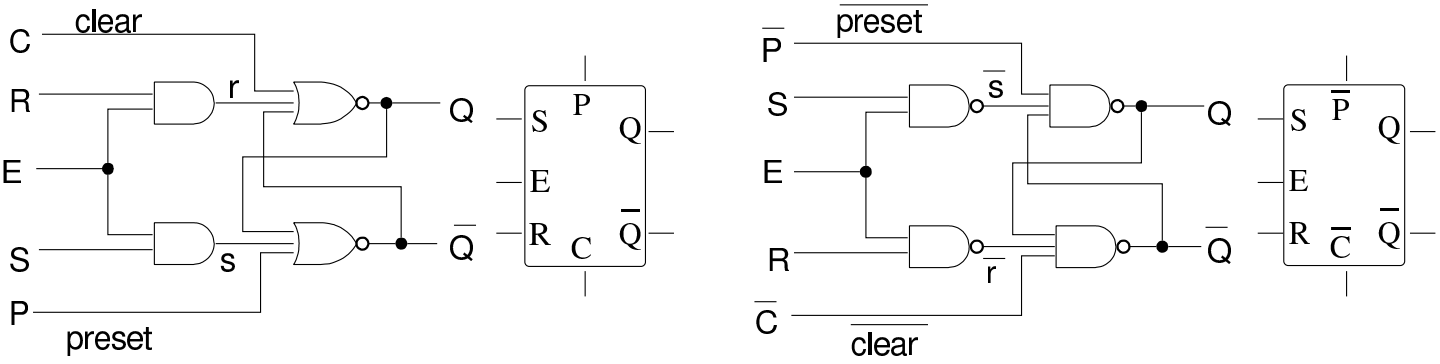
Figure 5.15. Truth table for a gated SR latch, limited to the most significant cases.



E	S	R	Q	$\overline{Q}$
0	x	x	*unchanged*	
_/‾‾_	0	0	*unchanged*	
_/‾‾_	0	1	0	1
_/‾‾_	1	0	1	0
_/‾‾_	1	1	*not allowed!*	
⋮	⋮	⋮		
1	‾_	‾_	*not allowed!*	
‾_	1	1	*not allowed!*	

When a gated SR latch is turned on, the initial output status is undetermined. To be able to initialize the circuit it is necessary to extend the input ports of the internal SR latch, as the following figure shows. It should be noticed that, depending on the internal SR latch implementation, the initialization signal might be asserted high or low.

Figure 5.16. Gated SR latch with initialization inputs: when the input $\boldsymbol{P}$ ($\overline{\boldsymbol{P}}$) input is asserted (preset), the $\boldsymbol{Q}$ output activation is forced; when the $\boldsymbol{C}$ ($\overline{\boldsymbol{P}}$) input is asserted (clear), the $\overline{\boldsymbol{Q}}$ output activation is forced (clearing the other output port).



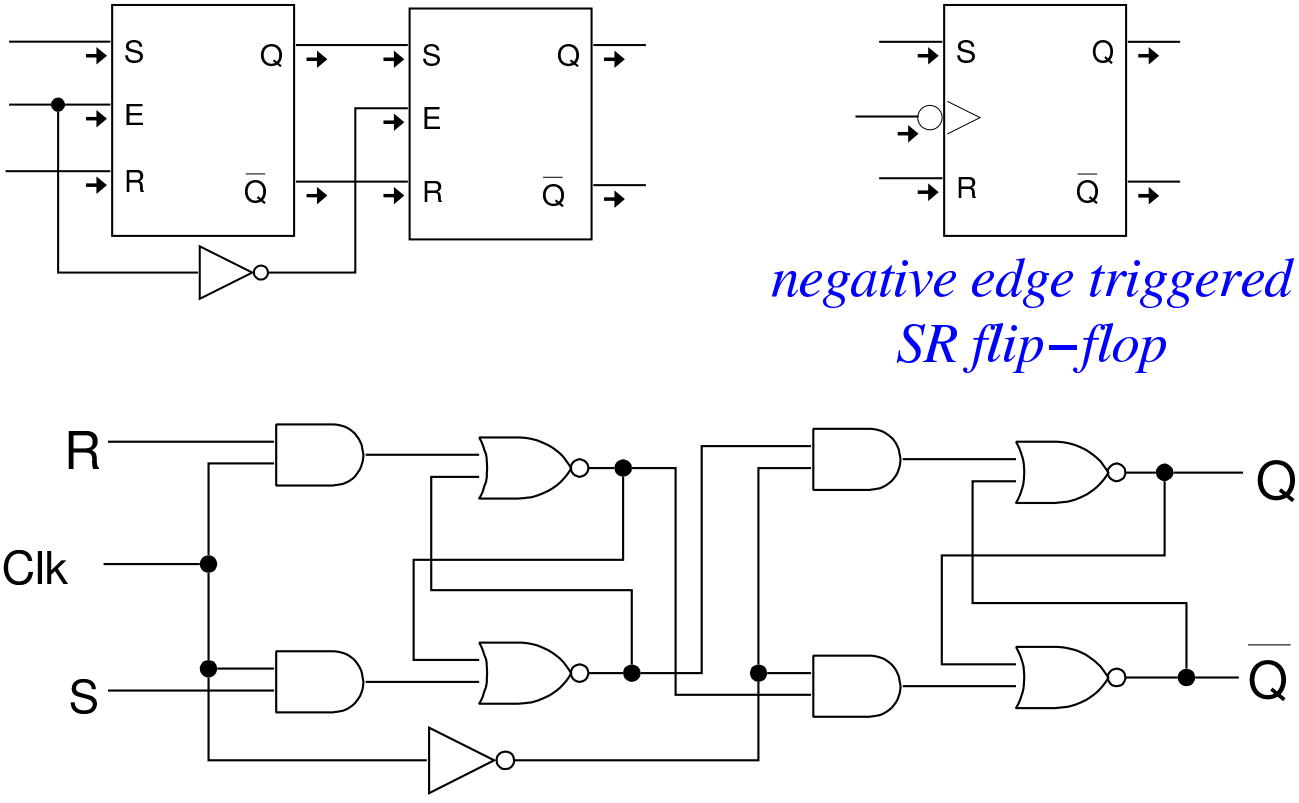
5.6 SR flip-flop

The gated SR latch has an enable input that, when asserted, allows to receive set and reset assertion from the other inputs, but if the enable input is to be used as a way to update the latch in a precise moment, the enable signal must be short enough to avoid that any change can happen in the meantime. To resolve properly the problem, two gated SR latches can be connected, one after the other as ‘master’ and ‘slave’, inverting the enable input to the second one: this way the output of the whole circuit

is updated when the enable signal passes from high to low, that is the ***negative edge***: this kind of latch is a ***flip-flop*** and it might be triggered with a negative or a positive edge, depending on the implementation.

Figure 5.17. Positive and negative edges of a signal.

Figure 5.18. Negative edge triggered SR flip-flop: when the enable signal goes from 1 to 0, the flip-flop is updated. The circuit is shown starting from the block diagram and then with the common gate implementations. It should be noticed that the enable input is now called 'clock' and it is usually shown as a triangle.



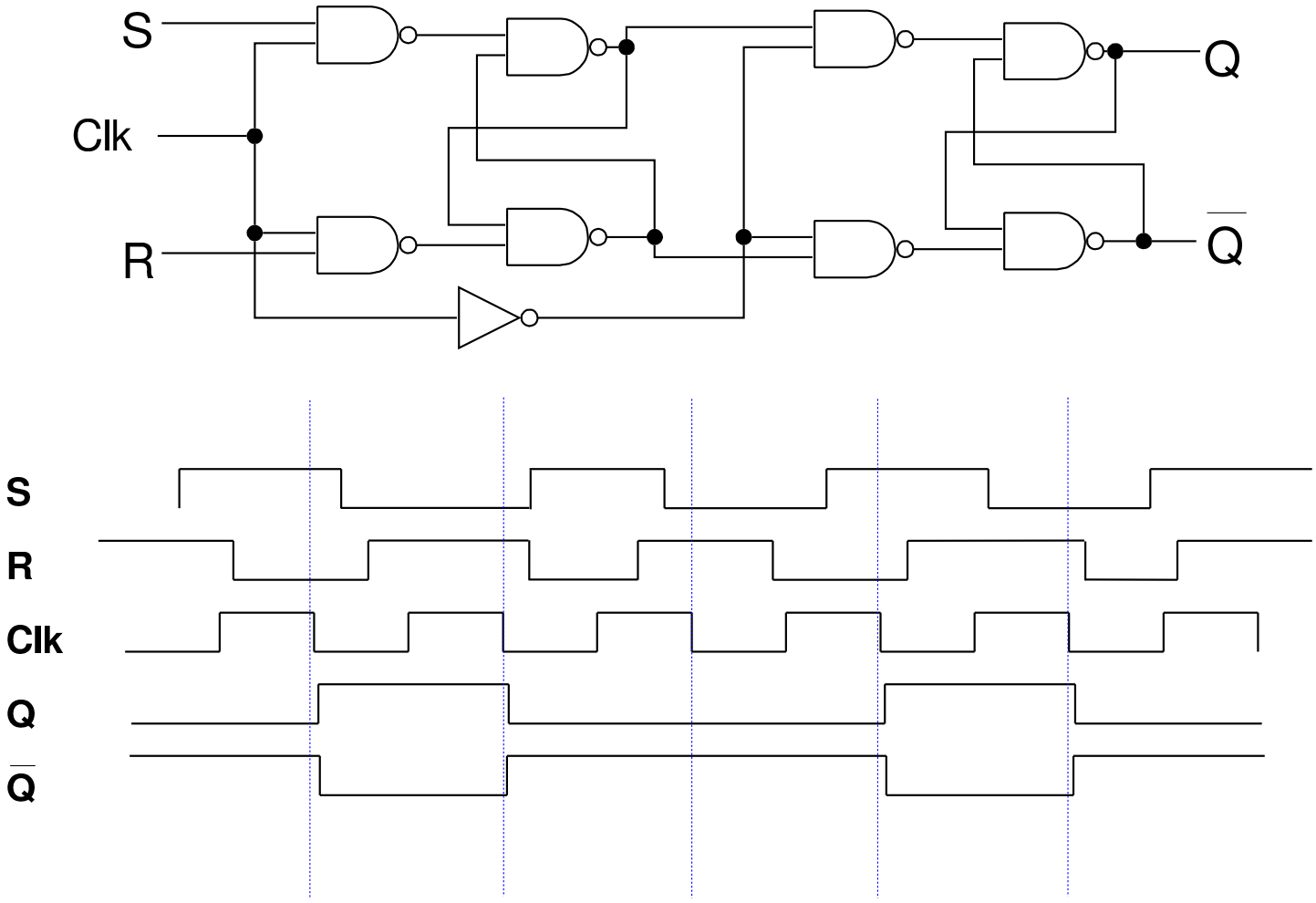
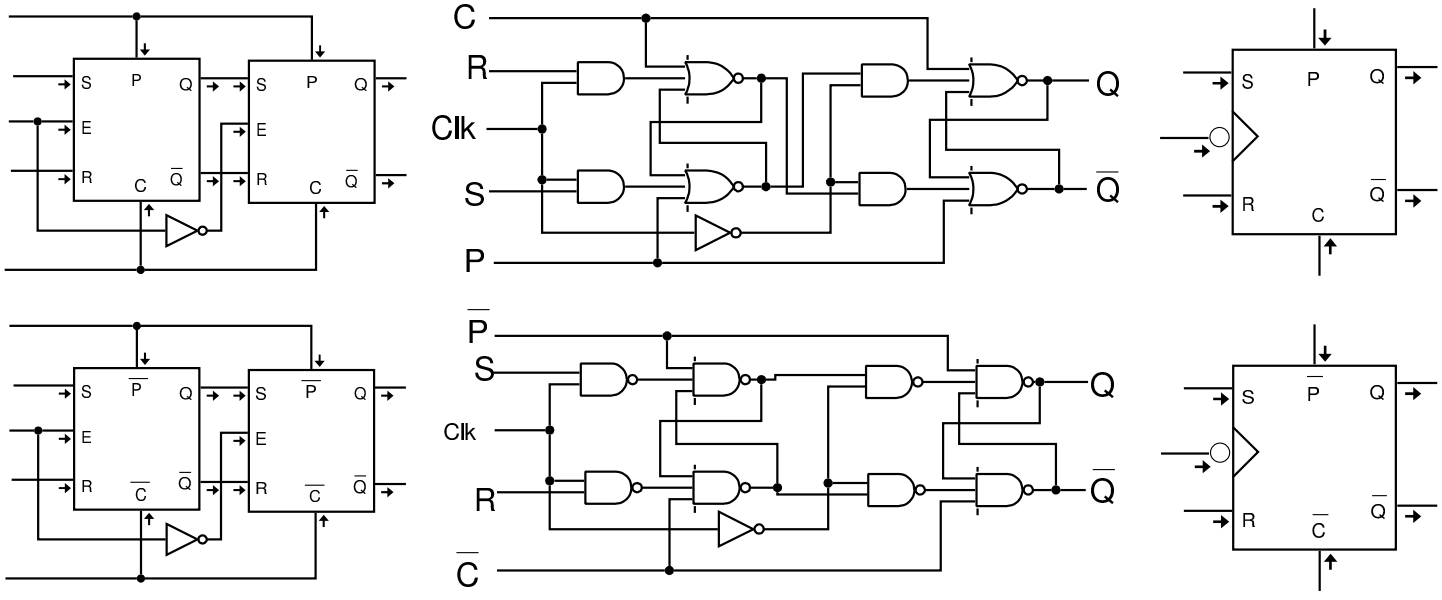


Figure 5.19. Negative edge triggered SR flip-flop, with initialization inputs.



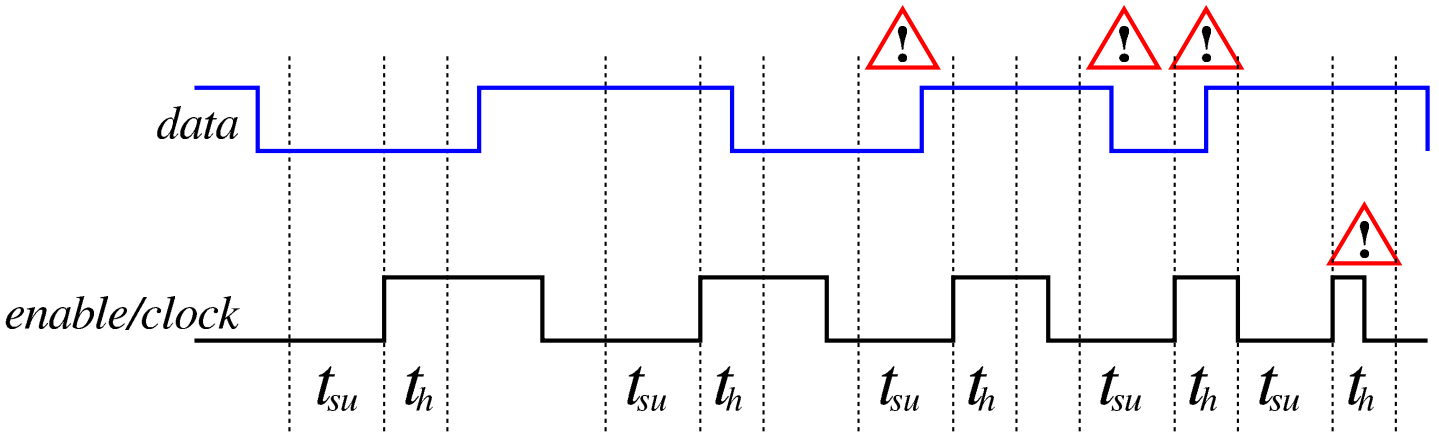
Please notice that the SR flip-flop (positive or negative edge) is never used; however, the gated latches connected as master and slave might be used to build more specific

flip-flops. Furthermore, instead of a hypothetical SR flip-flop, the JK flip-flop is used (see section 5.10).

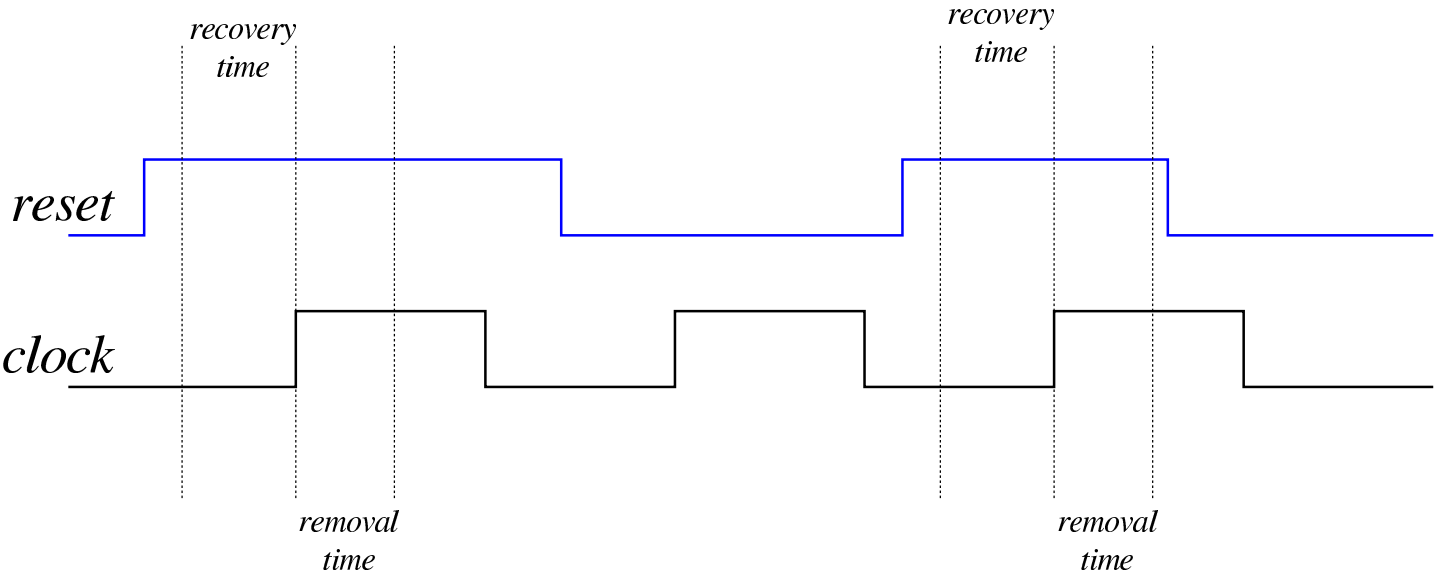
5.7 Time: setup/hold and recovery/removal

There are two significant time intervals related to the synchronous components, which are the circuits that have a data input controlled by an enable or clock input line. These time intervals are the ***setup time***, known with the label t_{su}, and the ***hold time***, known with the label t_h.

Figure 5.20. Example of correct and wrong conditions, concerning the setup time (t_{su}) and the hold time (t_h) constraints.



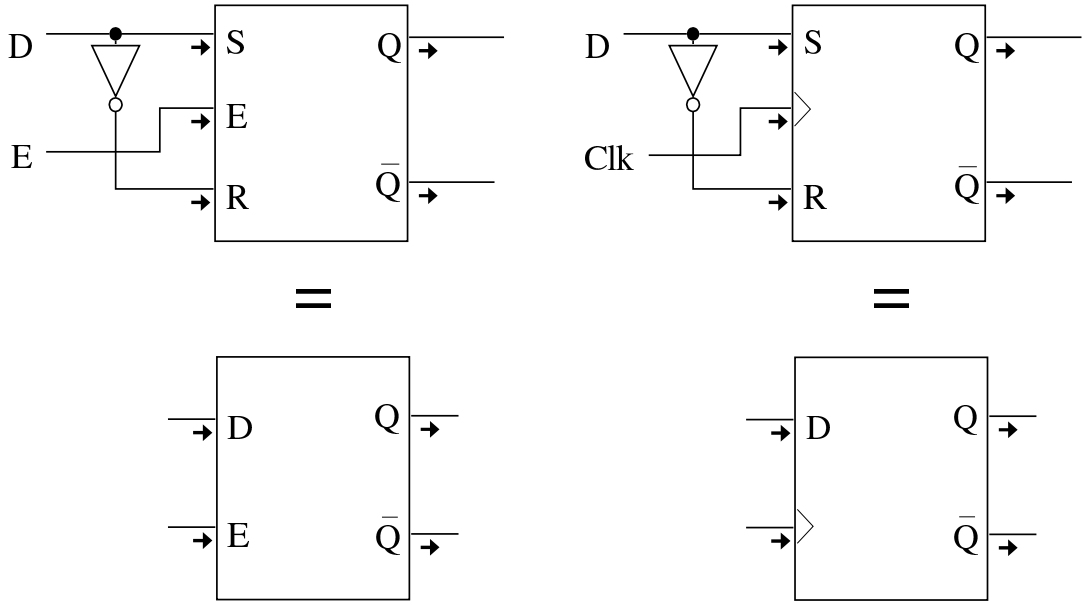
In different context, when it should be emphasized that the input data is asynchronous, other names are used: ***recovery time*** and ***removal time***.



5.8 D latch and D flip-flop

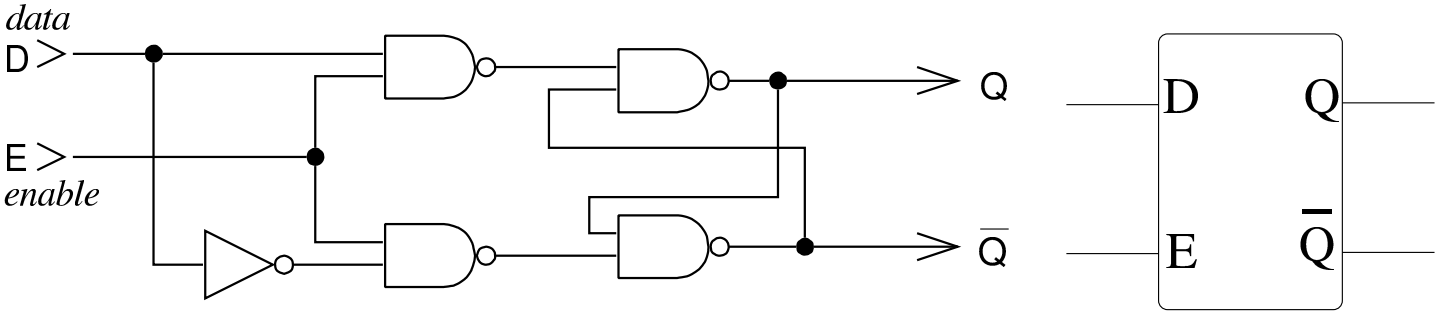
A gated SR latch or an edge triggered SR flip-flop can be connected so that the ***R*** input is the equal to the inverted ***S***. This way the single input line can be called ***D***, for 'data', and the new circuit become a D latch or a D flip-flop.

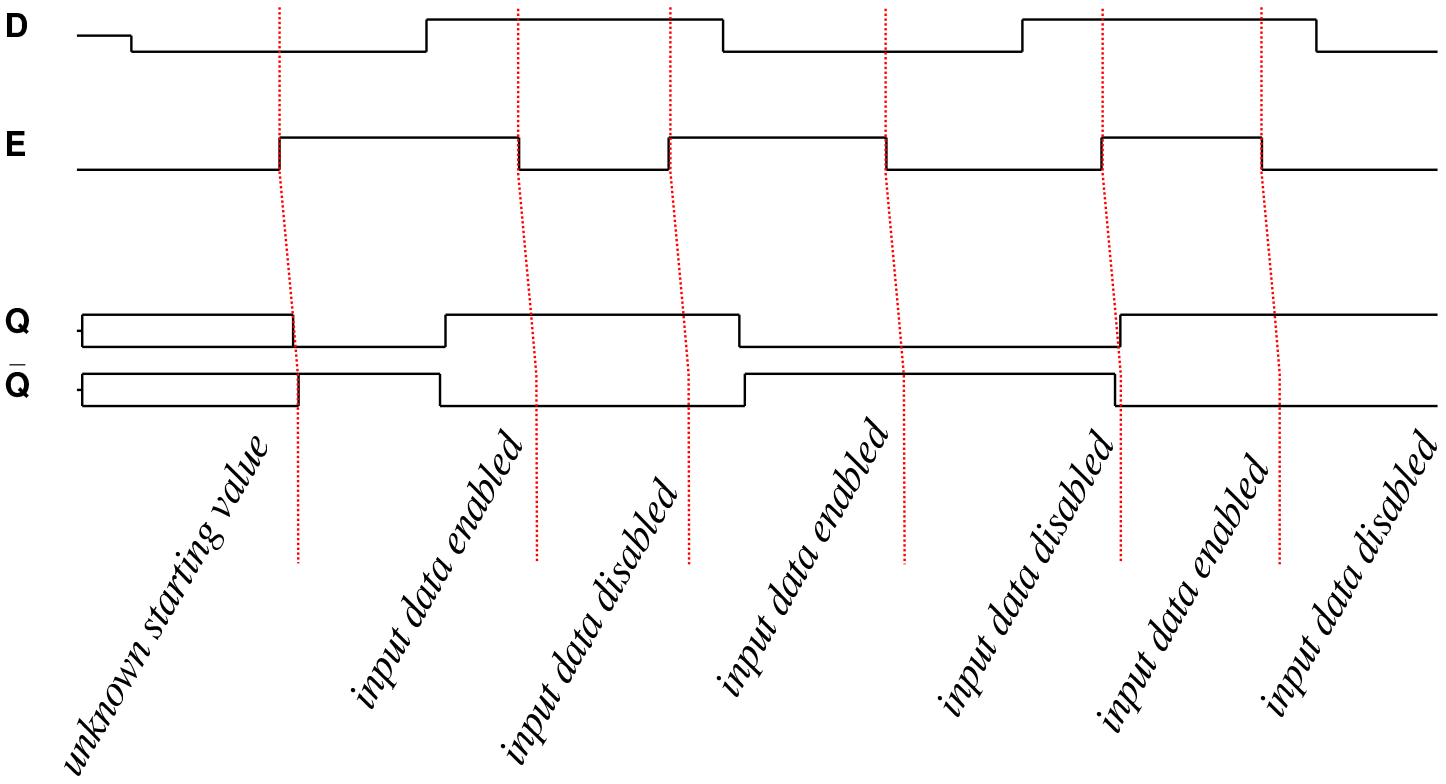
Figure 5.22. D latch and D flip-flop obtained from a gated SR latch or an edge triggered SR flip-flop.



As for the gated SR latch and the SR flip-flop, the same consideration about the setup and hold time apply to the 'D' variation. But the D latch do not have invalid combination for the input data that come form a single line.

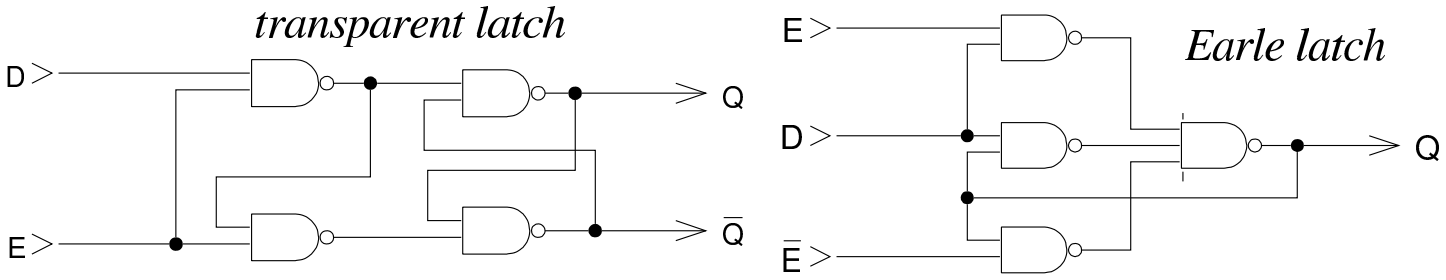
Figure 5.23. D latch made with NAND gates. When the ***E*** input is asserted, the circuit receives the data from the ***D***input and it reproduces the same value trough the ***Q*** output (inverting that value at the $\overline{Q}$ output).





The D latch can be implemented in more efficient ways, known as ***transparent latch*** and ***Earle latch***, which appear in the following figure. Please notice that the Earle latch has two complementary enable inputs, E and $\overline{E}$, which should be asserted and negated at the same time, to do their work properly.

Figure 5.24. The 'transparent latch' on the left and the 'Earle latch' on the right side.



The D flip-flop (edge triggered) can be implemented as master-slave, but there exists an alternative and more efficient circuit that triggers at the positive edge.

Figure 5.25. Master-slave D flip-flop and classical positive-edge-triggered D flip-flop. The timing diagram is related to the second version that triggers with positive clock edges.

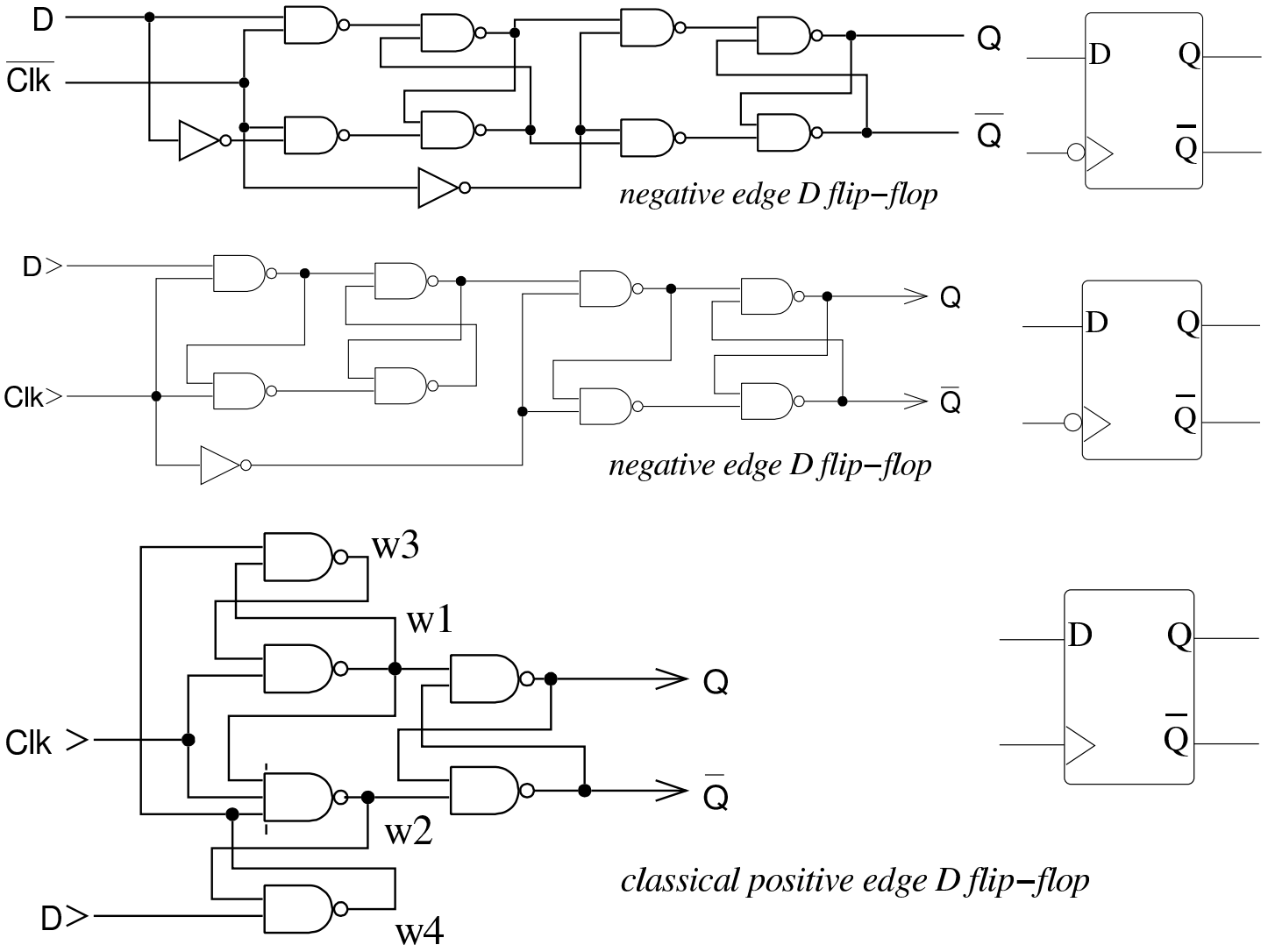
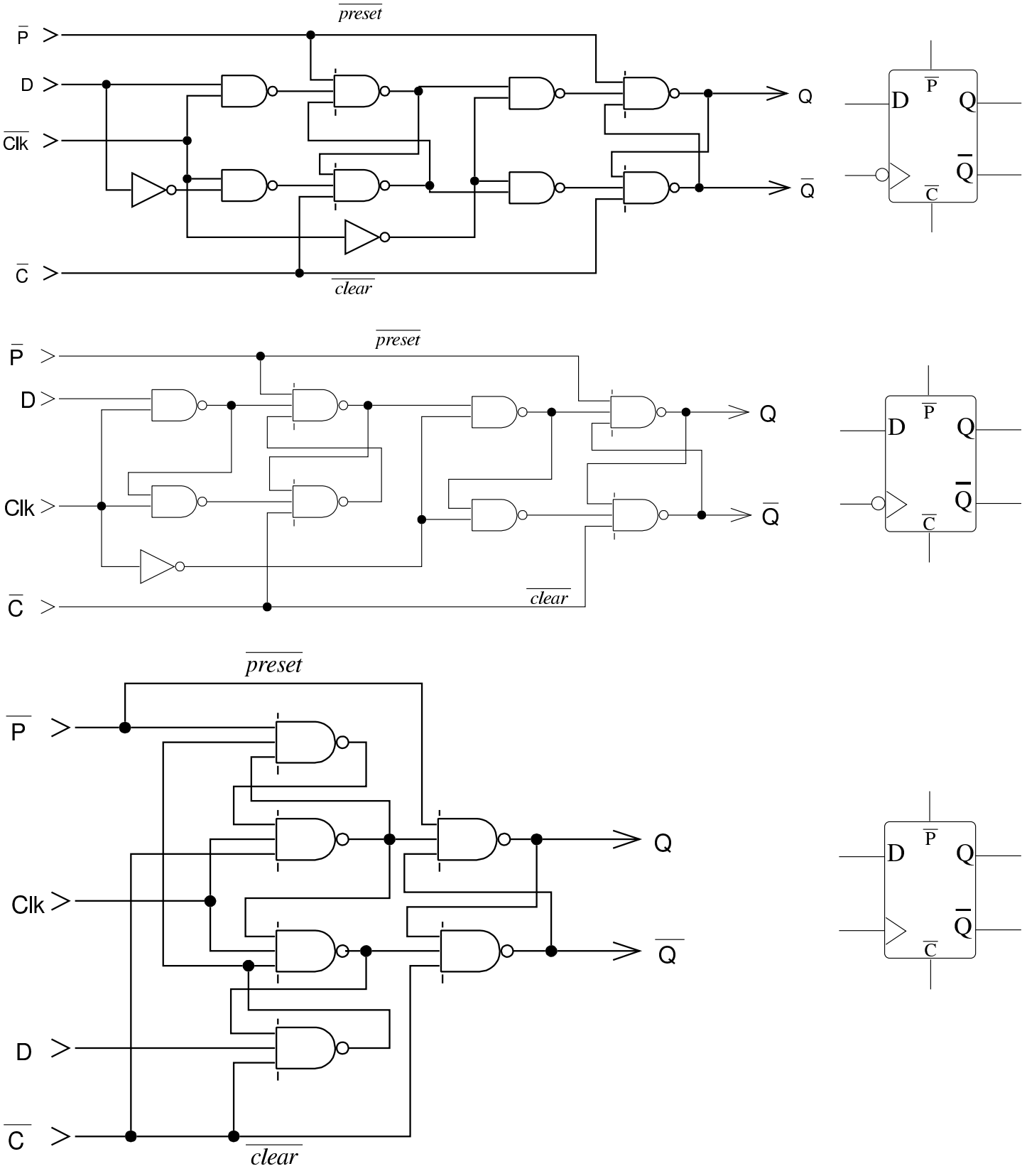


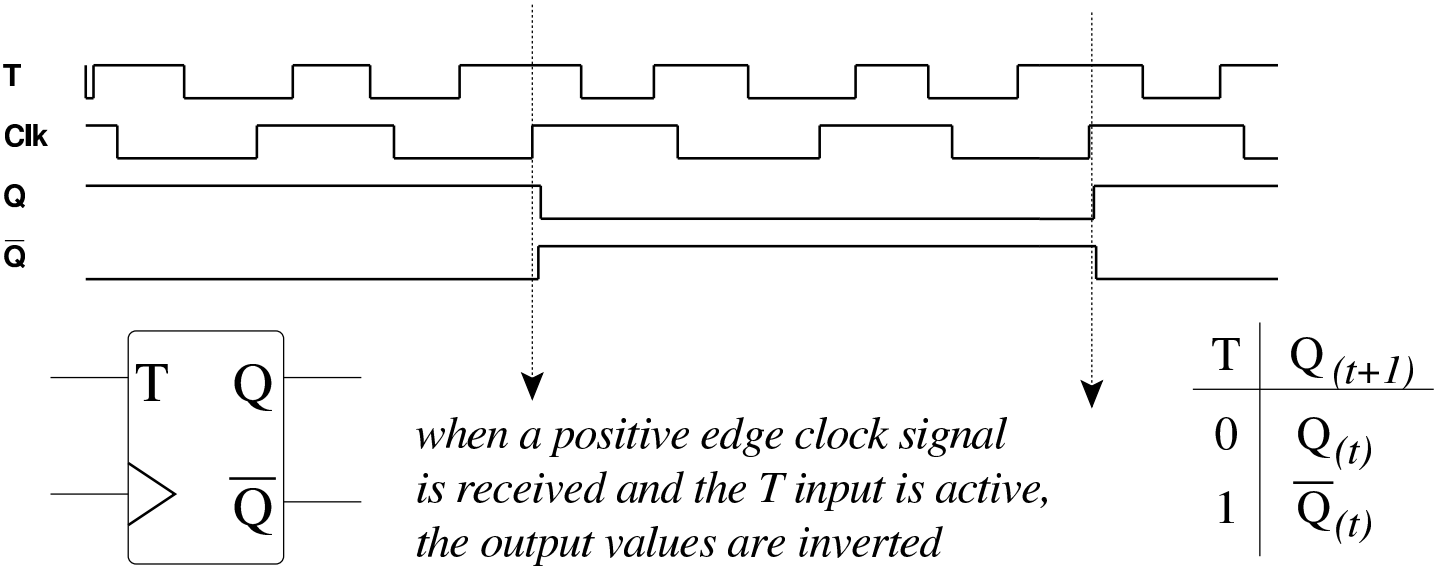
Figure 5.26. D flip-flops with initialization inputs.



5.9 T flip-flop

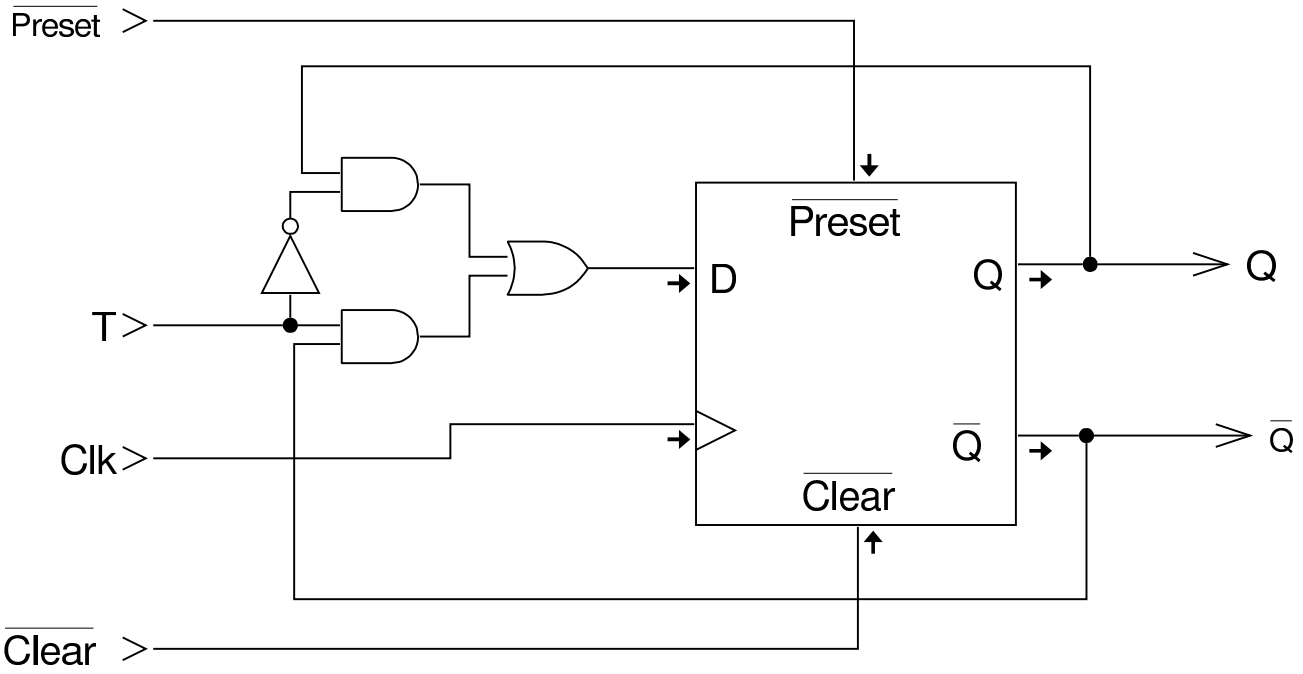
Extending a positive edge D flip-flop it is possible to obtain a T flip-flop, where the 'T' stands for 'toggle'. When the T flip-flop receives the clock edge and the ***T*** input is asserted, it inverts the output values. The truth table uses the notation $Q_{(t)}$ to indicate the ***Q*** output value at the time ***t*** and the notation $Q_{(t+1)}$ to indicate the ***Q*** output value at the next clock effective edge.

Figure 5.27. Positive edge T flip-flop: timing diagram and truth table.



The T flip-flop requires the clear or the preset input ports, because it is necessary to establish an initial value for the outputs. The following figure shows how to extend a D flip-flop with clear and preset inputs.

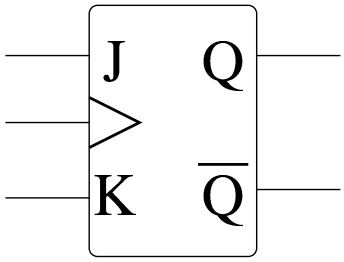
Figure 5.28. A T flip-flop starting from a D flip-flop, which includes clear and preset inputs.



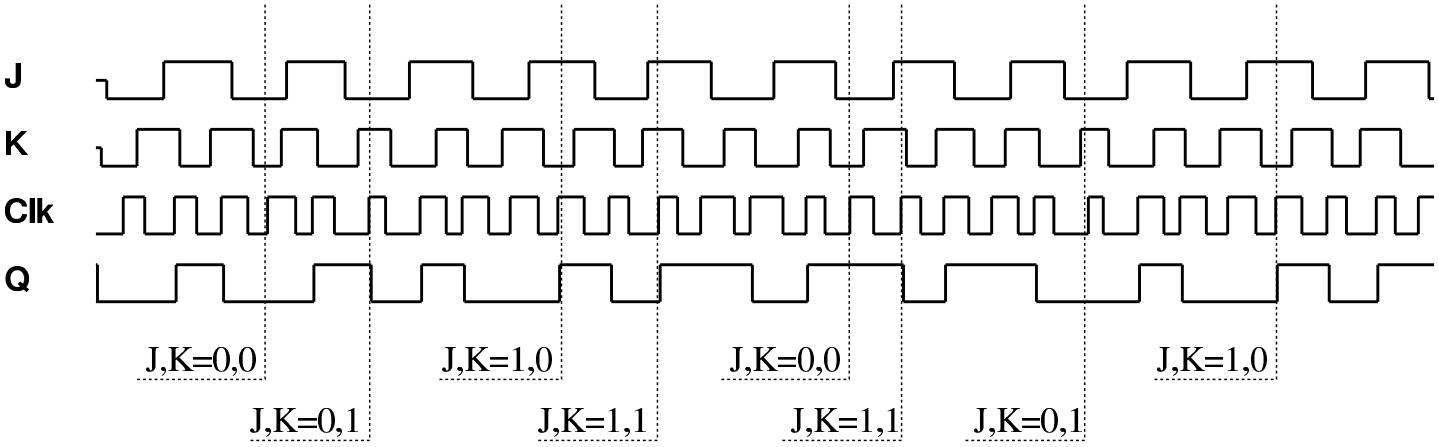
5.10 JK flip-flop

Another D flip-flop variation, recalling the SR latch, is the JK flip-flop. As for the T flip-flop, the truth table uses notations $\boldsymbol{Q}_{(t)}$ and $\boldsymbol{Q}_{(t+1)}$, to specify the $\boldsymbol{Q}$ value at the time $\boldsymbol{t}$ or at the next effective clock edge. In short, the JK flip-flop works like the SR latch, where $\boldsymbol{J}$ means 'set' and $\boldsymbol{K}$ means 'reset', but when both $\boldsymbol{J}$ and $\boldsymbol{K}$ inputs are asserted, the output values are just reversed, like it happens with the T flip-flop.

Figure 5.29. Positive edge JK flip-flop and truth table.

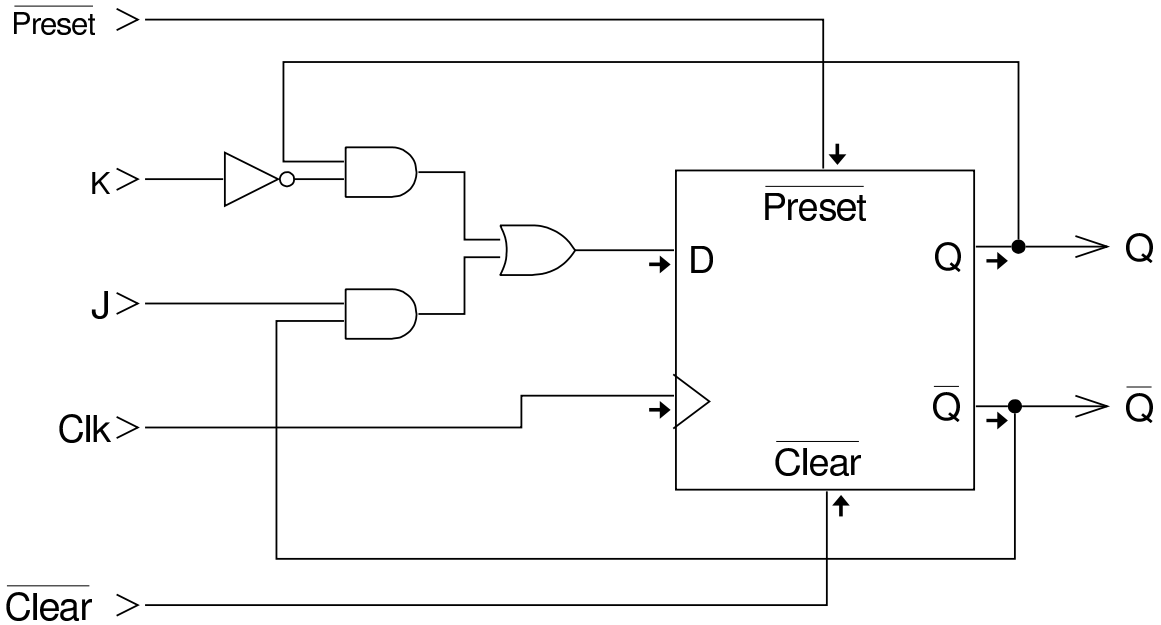


J	K	$Q_{(t+1)}$	
0	0	$Q_{(t)}$	*unchanged*
0	1	0	*reset*
1	0	1	*set*
1	1	$\overline{Q}_{(t)}$	*output inversion*



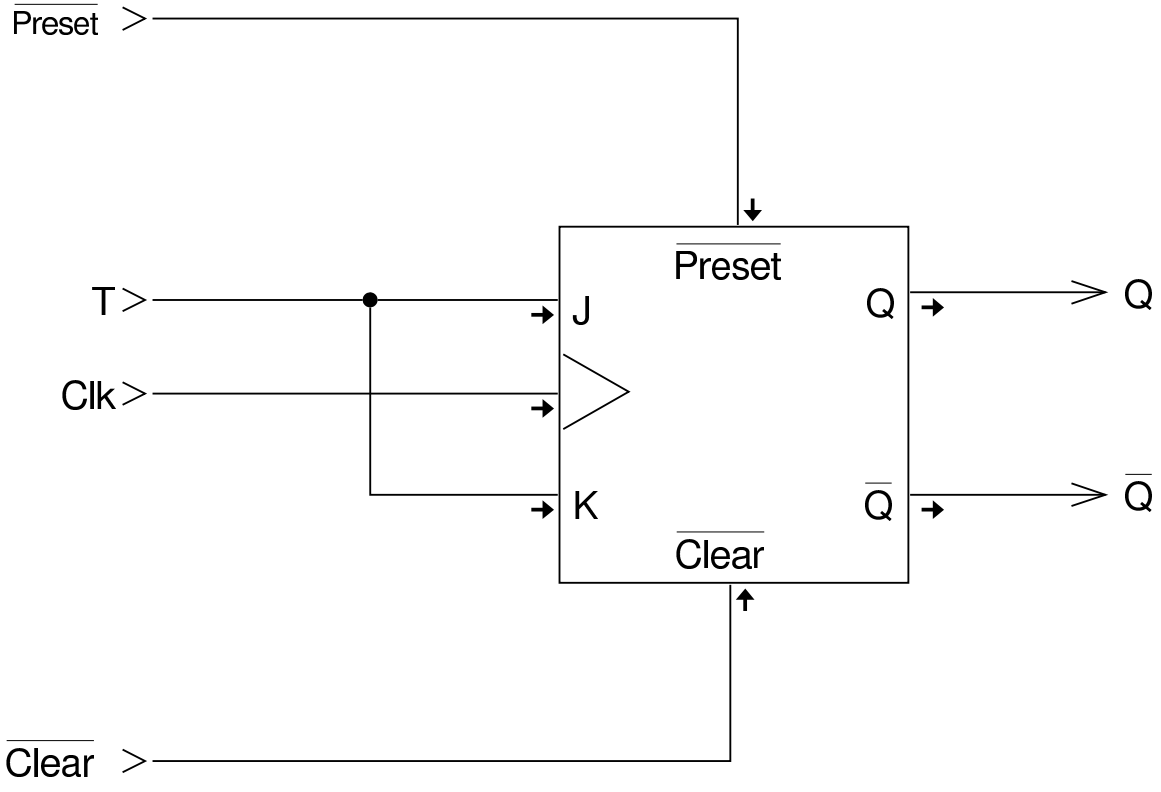
The JK flip-flop might require the clear or the preset input ports, but it is not strictly necessary as for the T flip-flop. Anyway, the following figure shows how to extend a D flip-flop with clear and preset inputs.

Figure 5.30. JK flip-flop made as an extended D flip-flop with clear and preset inputs.



It should be noticed that T flip-flop is just a JK one with the inputs ***J*** and ***K*** connected together.

Figure 5.31. JK flip-flop adapted to work as a T one.

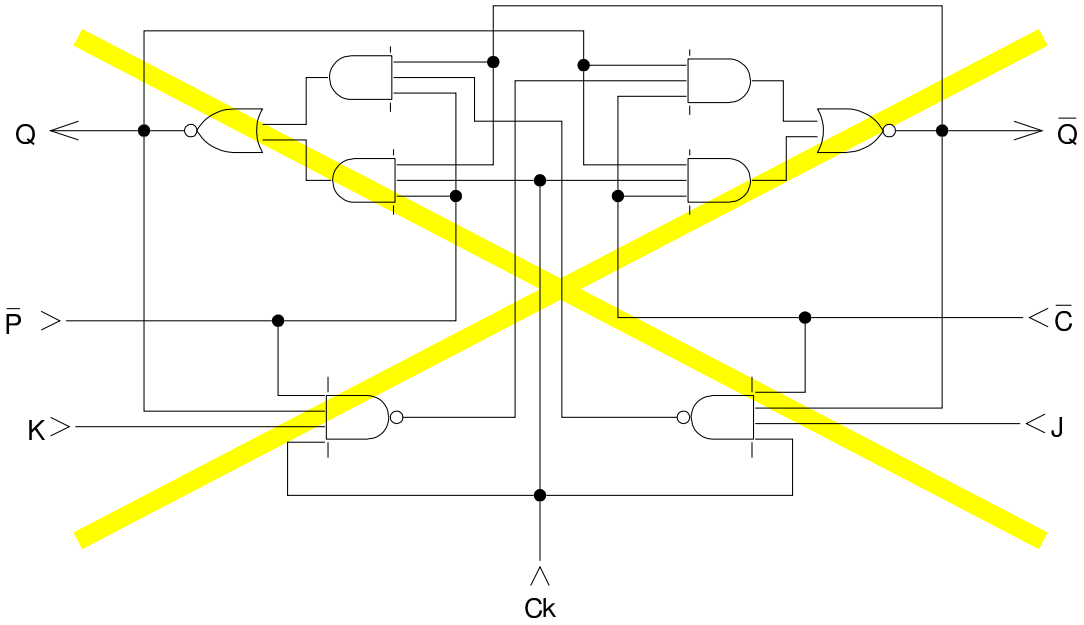


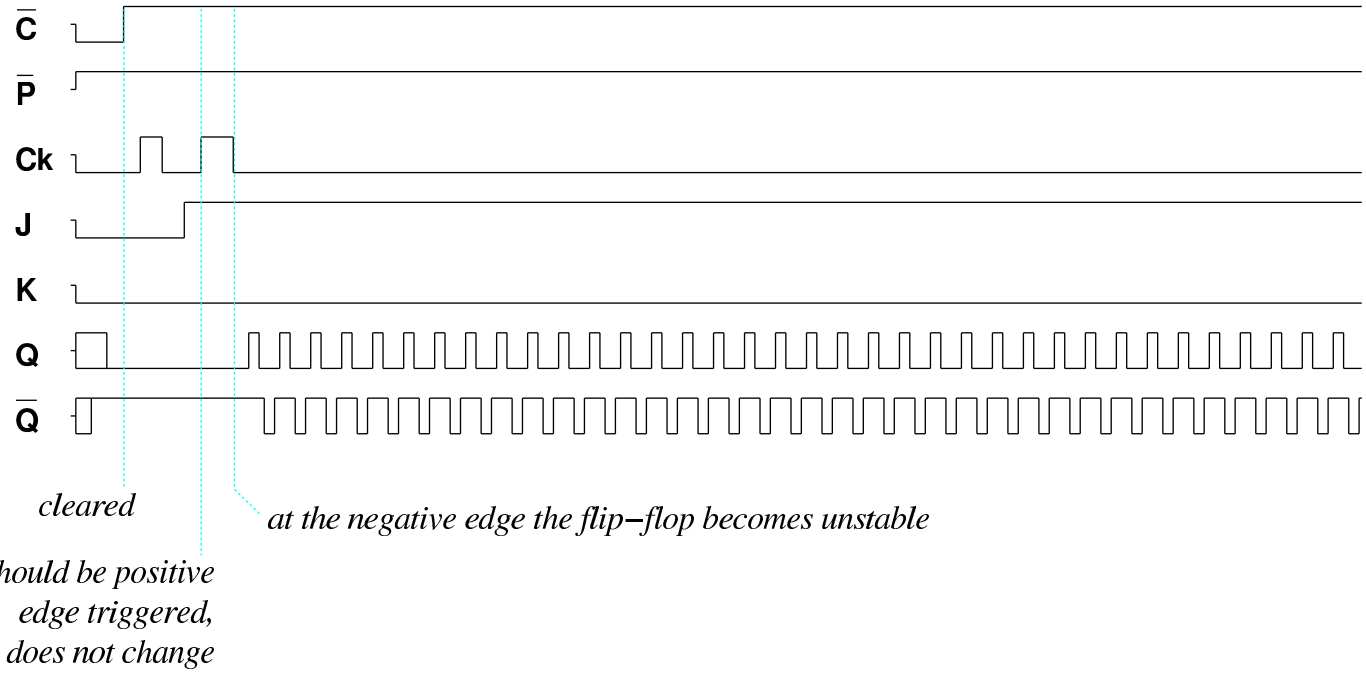
5.11 Troublesome JK flip-flops

Electronic flip-flops (D, JK, T) are currently produced with a specific technology that is not easily translated into common logic gates. The JK flip-flop obtained extending the D flip-flop works correctly, but there are other circuits inside the traditional documentation and data sheets that if reproduced with a simulator would not work as expected.

The following figure shows the circuit that can be found inside some original 7476 IC data sheets. This kind of circuit should be positive-edge triggered by the clock signal, but the timing diagram shows that it does not react this way and after the negative edge the circuit becomes unstable.

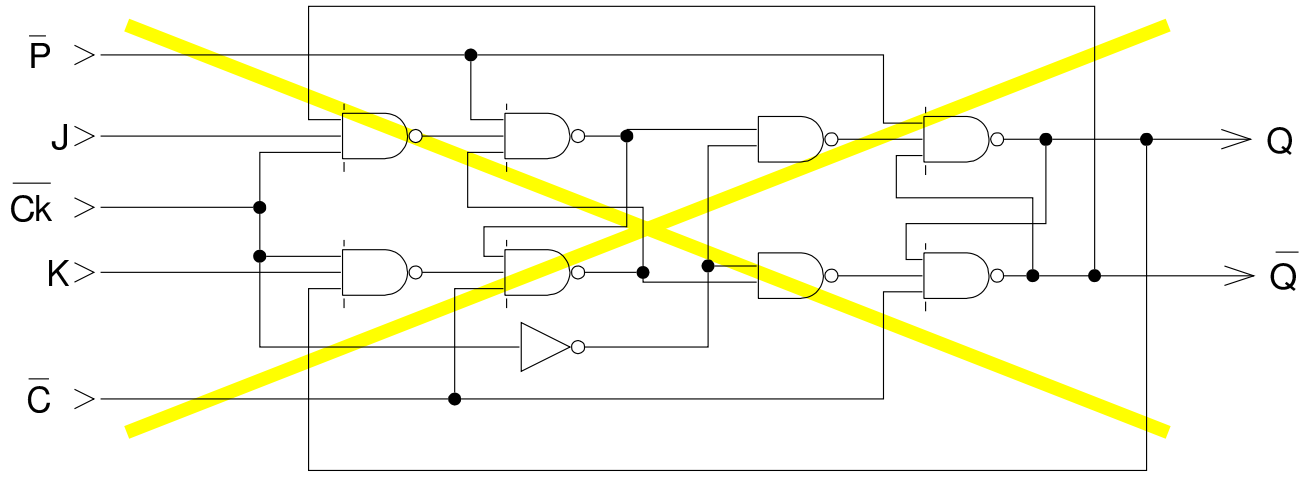
Figure 5.32. JK flip-flop circuit that does not work.

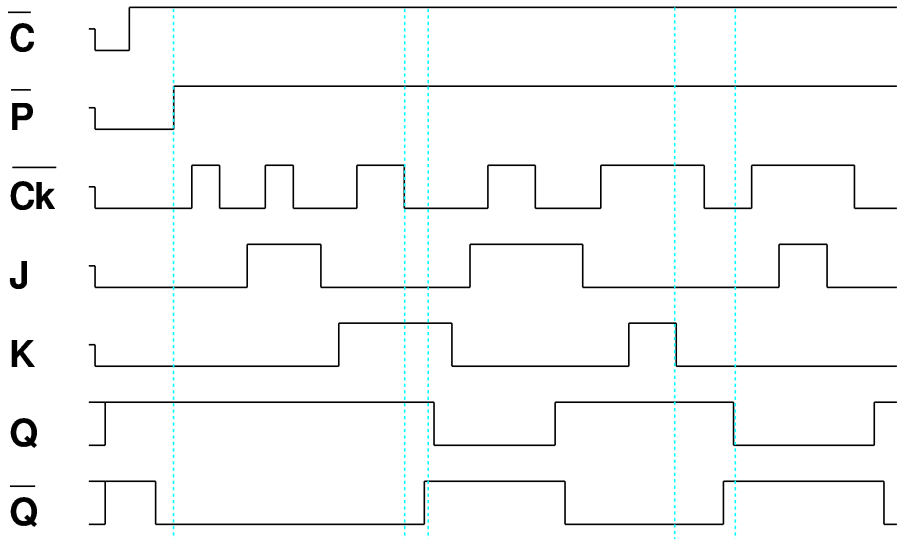




The following figure shows a master-slave JK flip-flop (negative-edge triggered) that is described in the traditional documentation. This circuit might work, but it does not completely as expected: if the ***J*** or ***K*** input signals are negated before the clock negative edge, when the clock negative edge actually comes they switch the flip-flop anyway.

Figure 5.33. Master-slave JK flip-flop.





cleared

negative edge

output updated

K goes down before the clock negative edge

the output is updated anyway

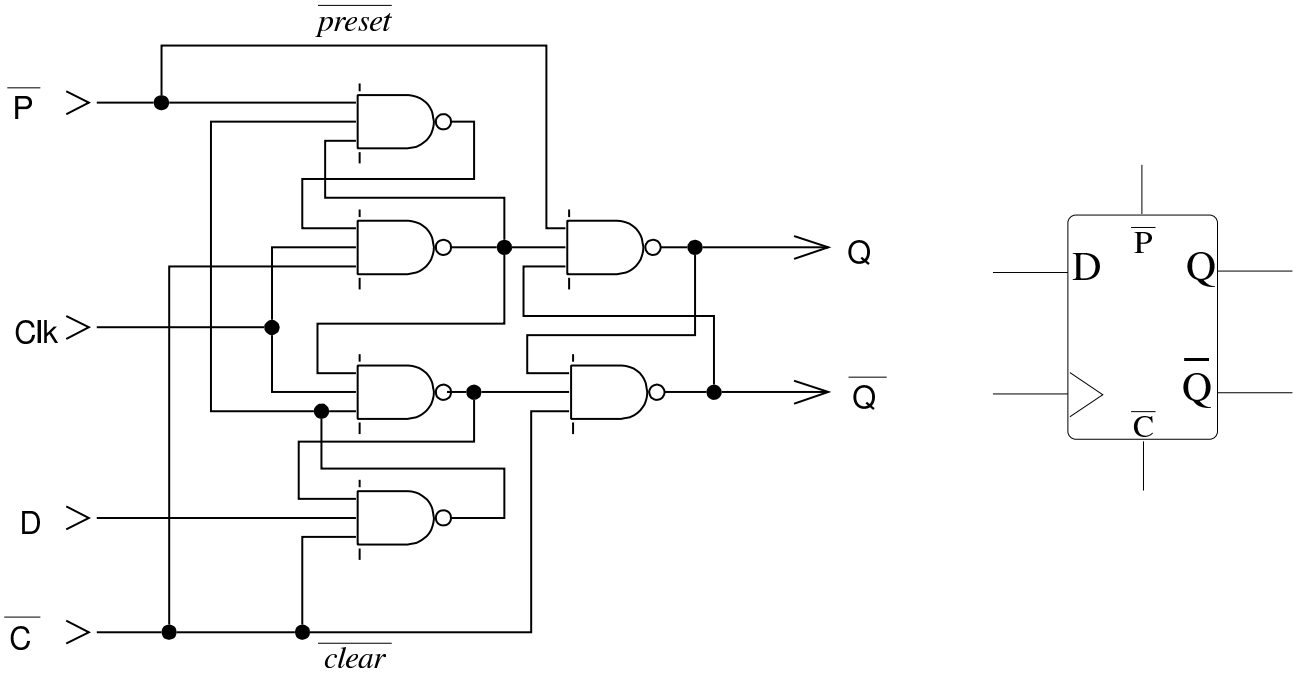
6.1 Simple data register 212
6.2 Shift register 216
6.3 Asynchronous counter with T flip-flop 217
6.4 Synchronous T flip-flop counters 219
6.5 Synchronous counters with D flip-flops 220
6.6 Synchronous counters with paralel load 224

counter 219 220 counter register 217 data register 212 register 211 shift register 216

A ***register*** is a flip-flop array (usually D flip-flops are used) by which it is possible to store binary values or to build a counter. It could be possible to build registers with gated latches, but the edge-triggered flip-flops are more reliable for the purpose.

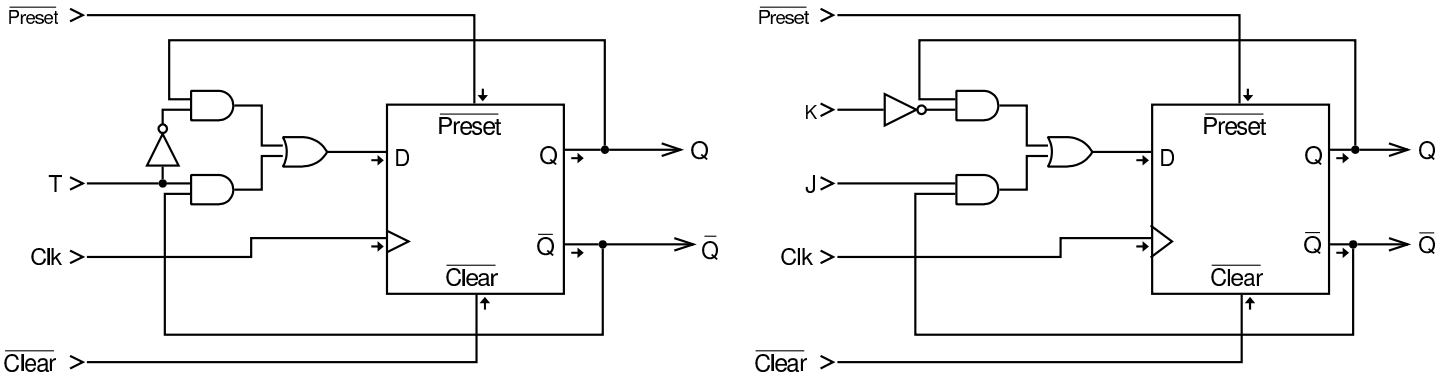
The following examples use D flip-flops following the classical implementation or derived from that kind of circuit. Therefore, these are positive-edge triggered flip-flops.

Figure 6.1. Classical D flip-flop.



"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

Figure 6.2. T and JK flip-flop, derived from the D flip-flop.

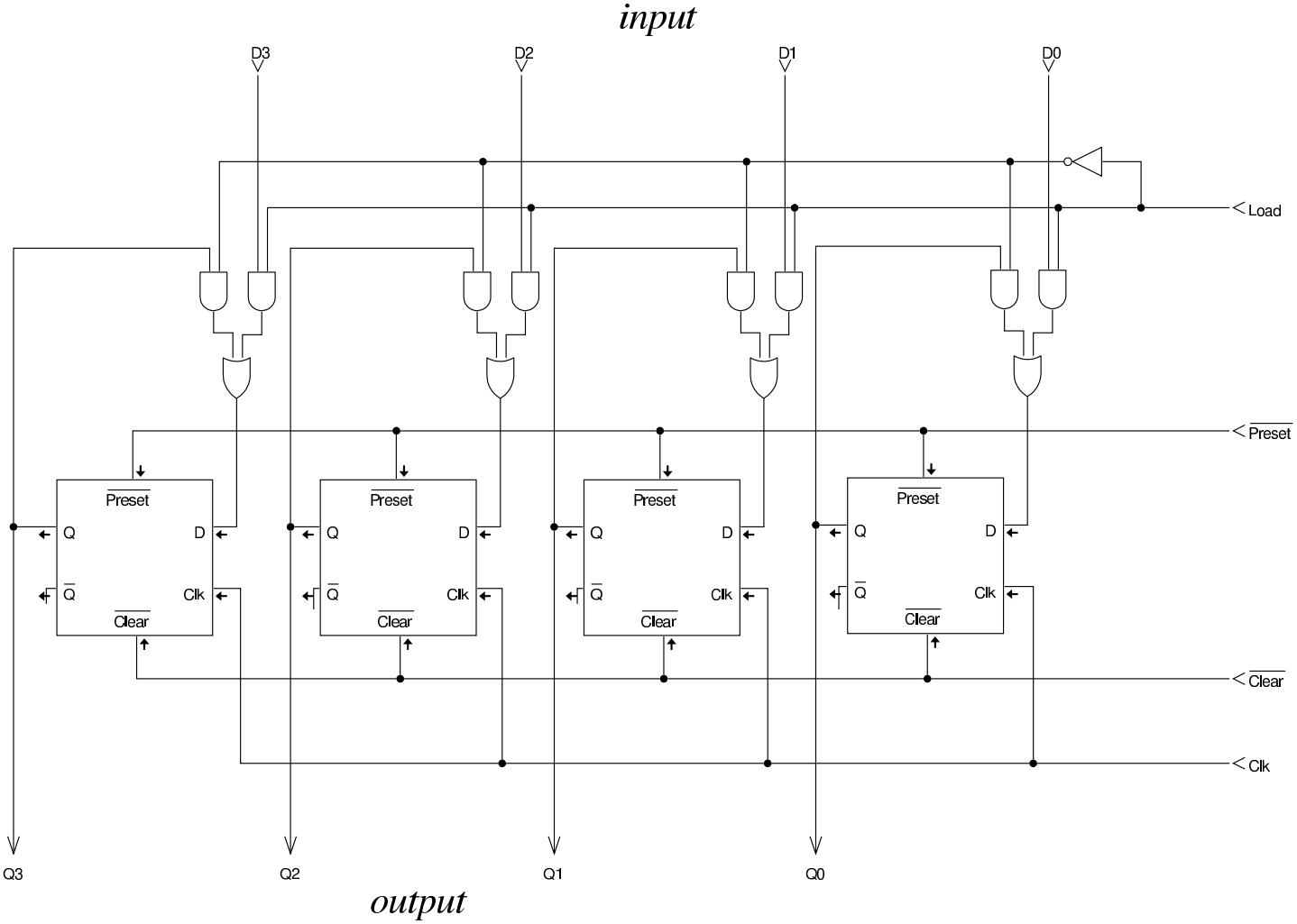


The figures appearing in the following sections might call the clock line either with the abbreviation ***Clk*** or ***Ck***, but the flip-flops are meant to be edge-triggered, even if there is not the standard triangular symbol for this purpose.

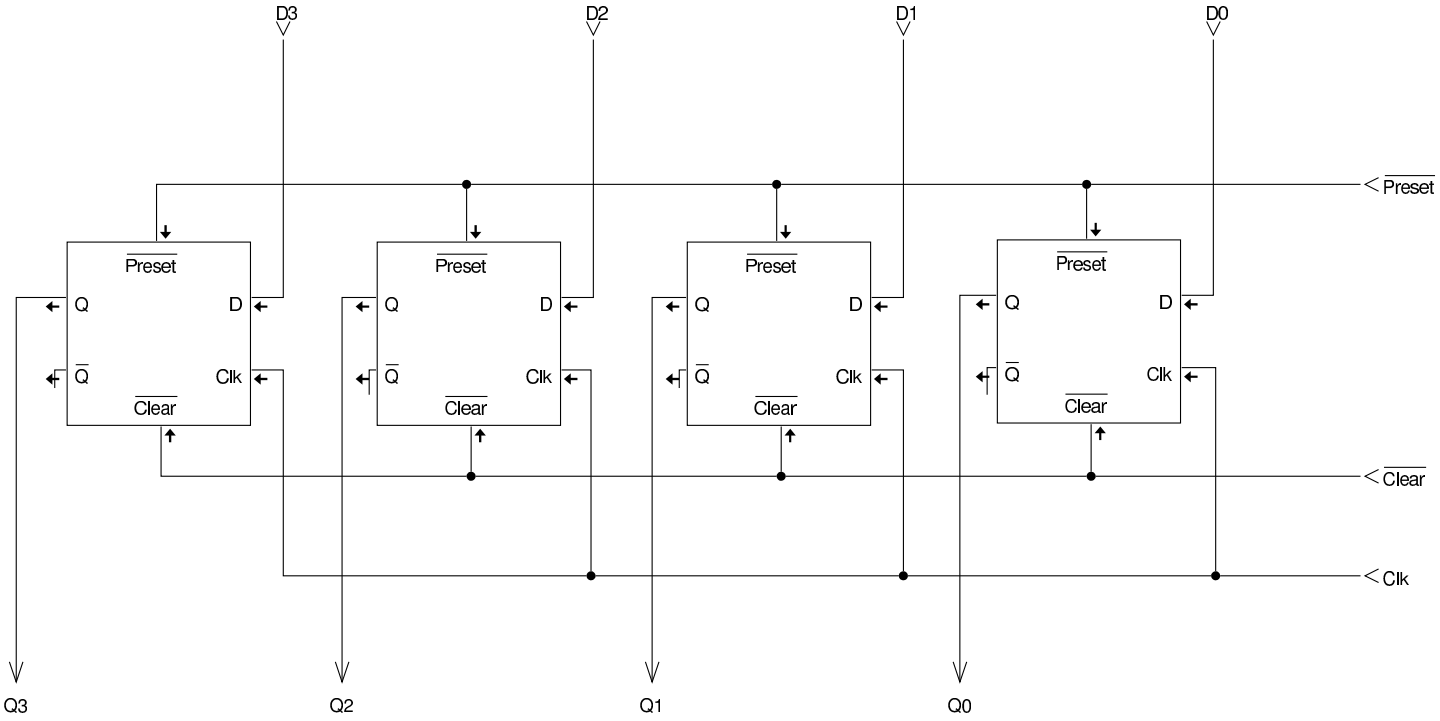
6.1 Simple data register

The simplest register is the one that can load a multi-bit value, storing it until the input lines are enabled again. In the following figure, the value stored by the register can be read from the output lines $\boldsymbol{Q}_{3..0}$ and a new value can be loaded from the input lines $\boldsymbol{D}_{3..0}$. The input value is loaded only when the input ***Load*** is asserted, otherwise the flip-flops just reload the previous value at every clock positive edge.

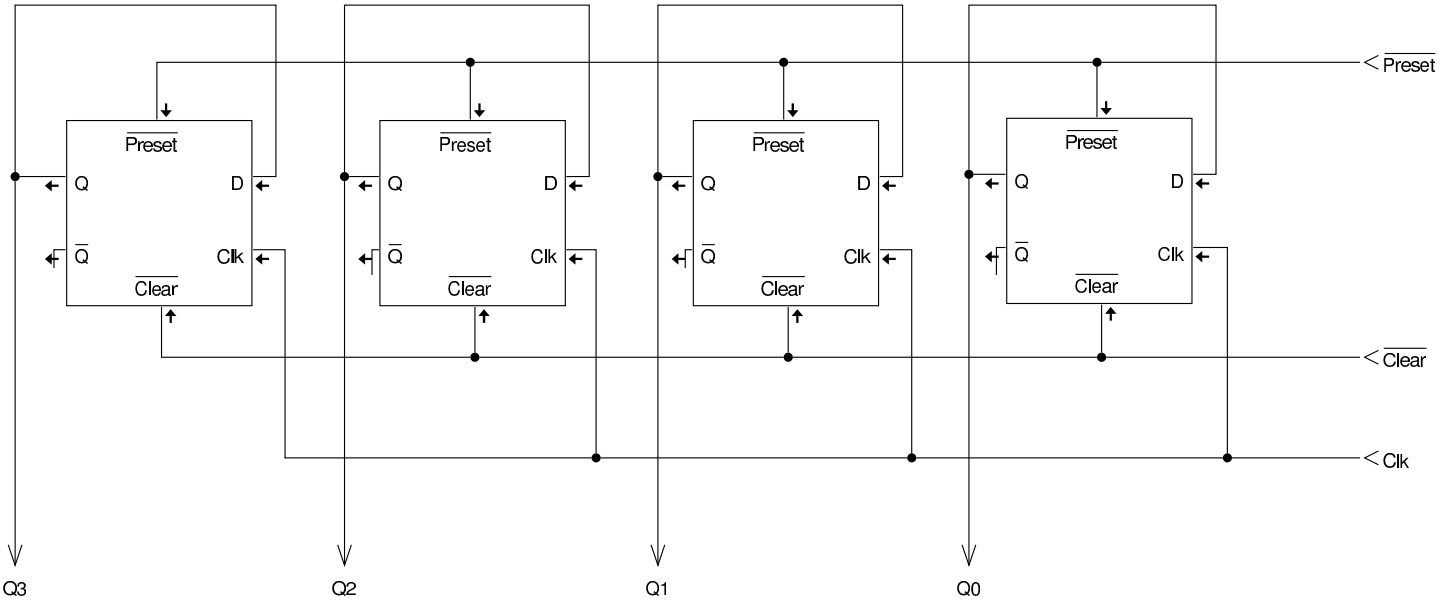
Figure 6.3. 4-bit register: when the ***Load*** input line is asserted and a positive edge clock is reached, the register loads the value available from the input lines $\boldsymbol{I}_{3..0}$, otherwise the previous stored value is just kept as it is.



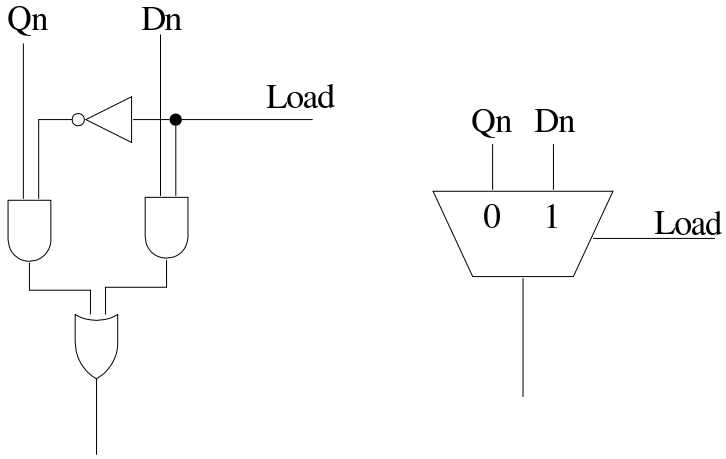
To allow a better understanding, the following figure shows how the register works when it is willing to receive the input data, removing the logic that switches the data sources:



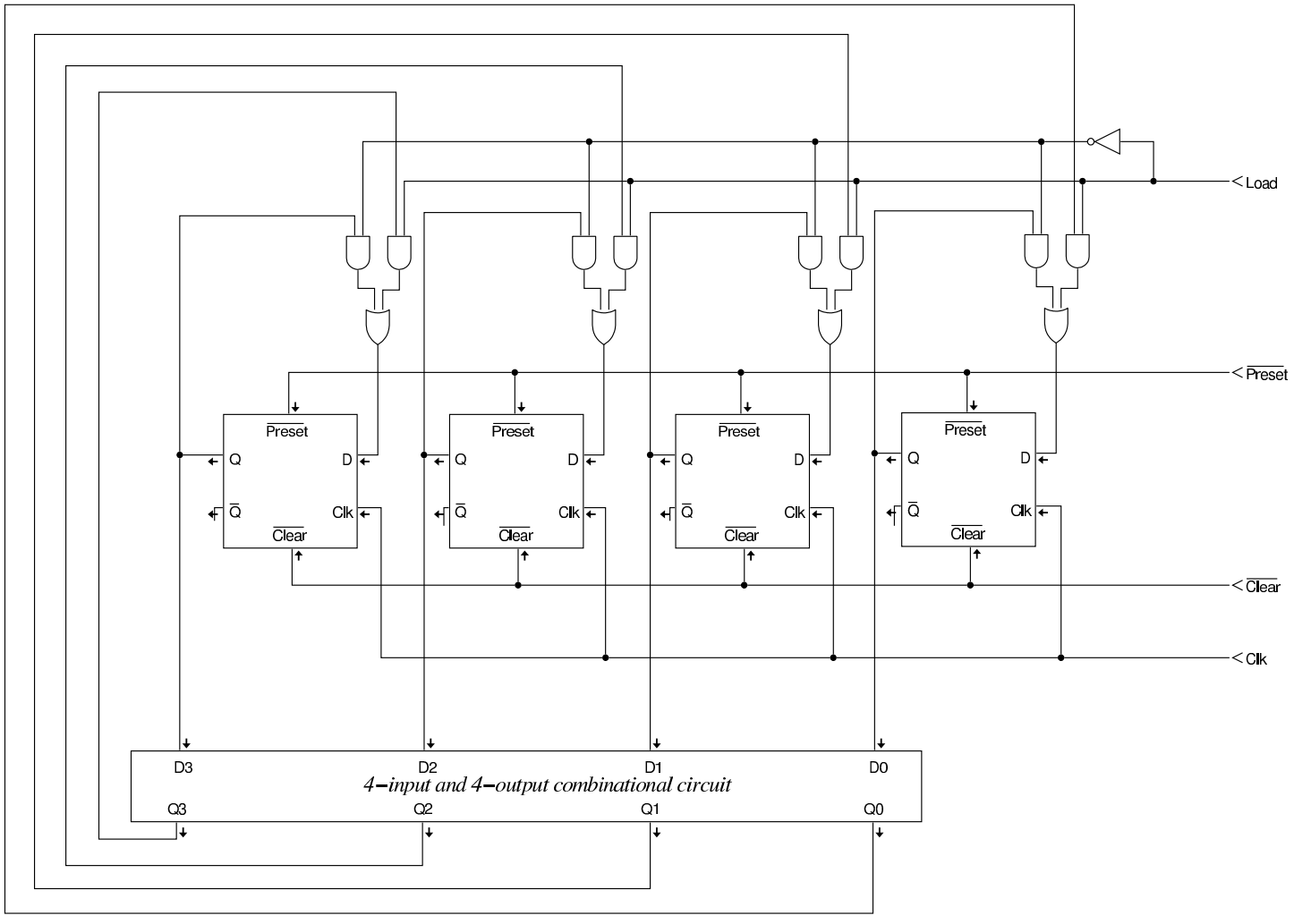
When the registers keeps the previous stored value, it works like the following figure:



I should be clear that the ***D*** flip-flop inputs are switched by a multiplexer, as the following figure can clarify:



The simple data register has an important feature: it can be updated starting from the value that it contains already, when the positive clock edge is received. The following figure shows a data registers that might load a new data from a combinational circuit, which calculate something on the current register value:



The register output is connected to a combinational circuit that transforms the data in some unspecified way. The combinational circuit then feeds the register when the

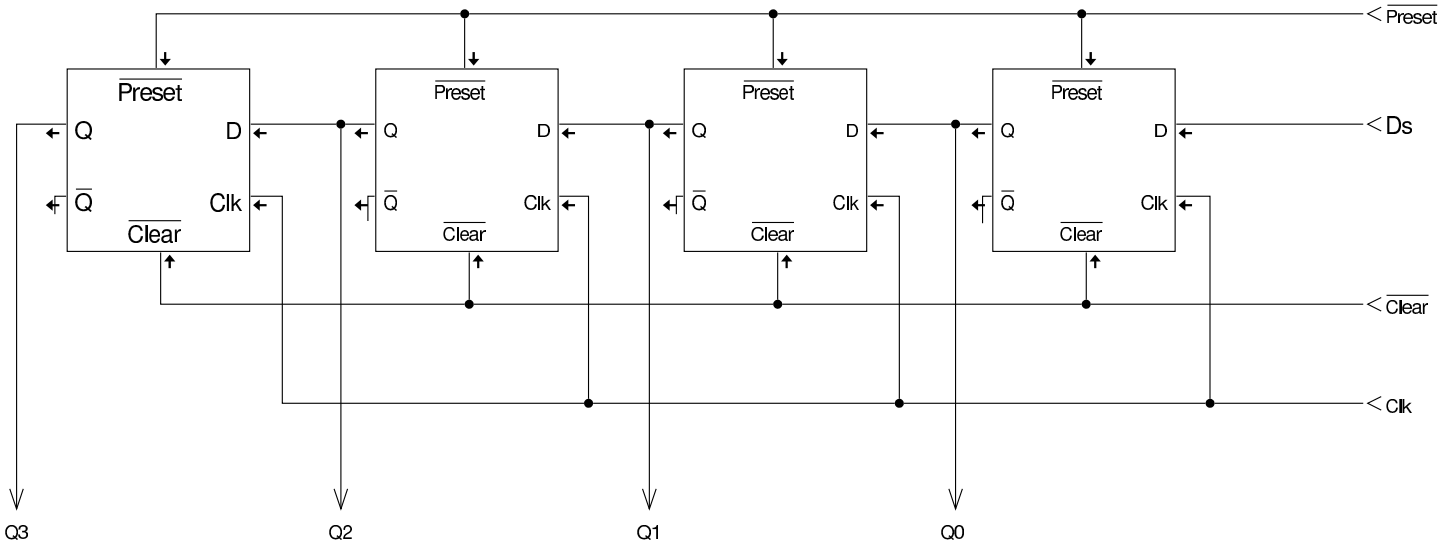
Load input is asserted and the clock reaches the positive edge. In practice this circuit does what could be formally written as ***x=f(x)***, or ***x<--f(x)*** if the RTN notation (*register transfer notation*) is used.

6.2 Shift register

«

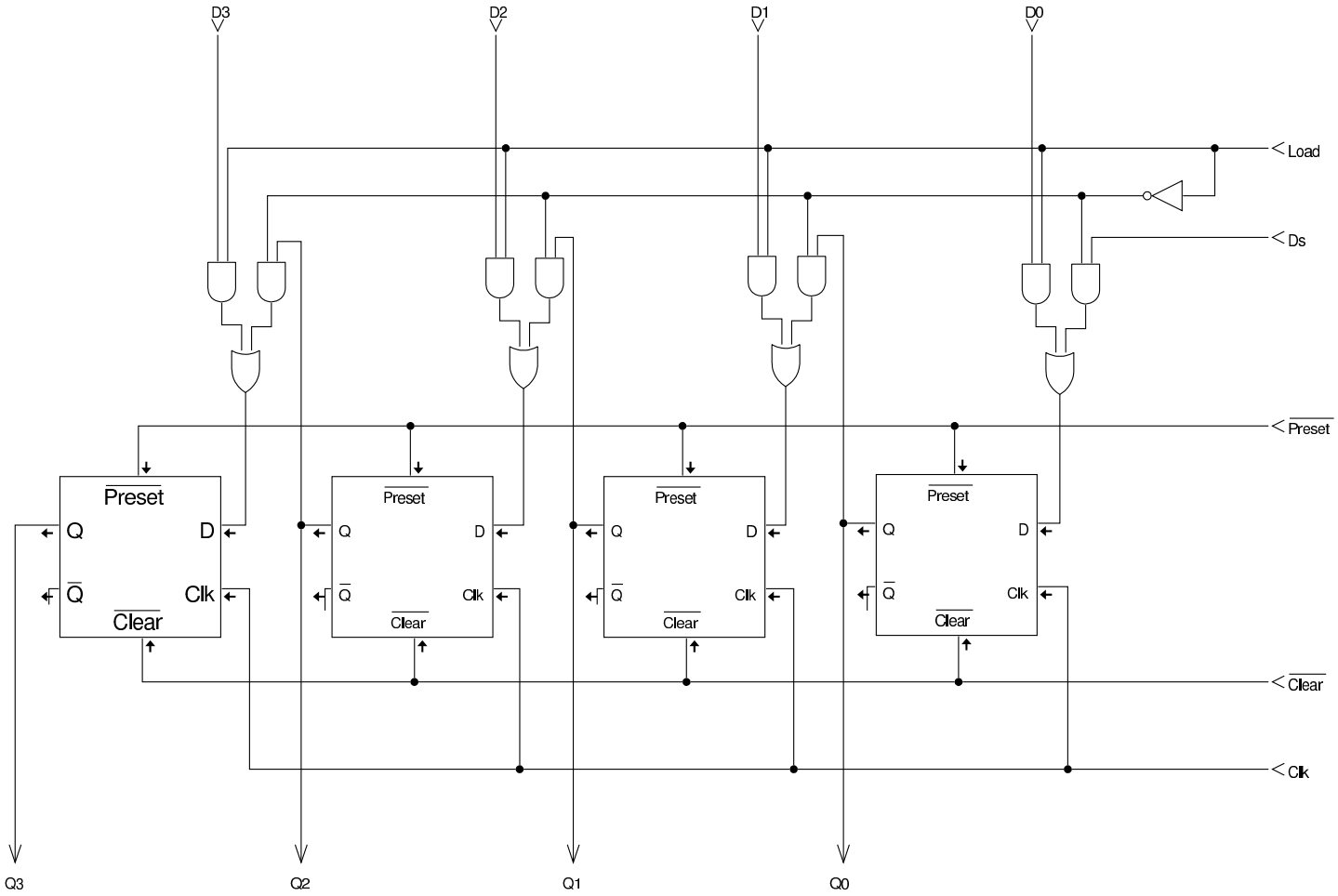
The following figure shows a register that can receive a single logic value from the input ***Ds***, when there is a positive edge clock signal. At this time, the first flip-flop on the right updates its value with the serial data obtained by the input ***Ds***, whereas the second flip-flop updates itself using the previous value from the first flip-flop and the same thing happens to the other flip-flops.

Figure 6.8. 4-bit shift register: when there is a positive edge clock signal, the input value is stored inside the first flip-flop on the right, whereas the second gets the old value from the first one, and so it continues up to the last flip-flop. Video: ogv *http://www.youtube.com/watch?v=I_ncbpQnCZ4* .



A shift register, like the one that appears in the previous figure, can be extended so that it can load a value. In that case a ***Load*** input can be added to switch the register flip-flops, like the following figure shows.

Figure 6.9. 4-bit shift register with paralel load: when the input ***Load*** is asserted and the clock signal reaches the positive edge, the register stores the value that is available at the input $\boldsymbol{D}_{3..0}$, otherwise, if the input ***Load*** is negated, each flip-flop is updated with the last value from the previous one.



6.3 Asynchronous counter with T flip-flop

The counters are registers that increment or decrement their binary value at every clock pulse; the asynchronous ones contains flip-flops that are not connected to the same clock signal. The easiest counter is made of T flip-flops, because this kind of flip-flop, when the ***T*** input is asserted, inverts the output value each time it receives a clock pulse. The following figures show the use of cascaded T flip-flops, which means that each clock input is driven by the output of a previous one: that is why these are asynchronous counters.

Figure 6.10. 4-bit ascending asynchronous counter: the next flip-flop is driven by the $\overline{Q}$ output from the previous one. Video: ogv *http://www.youtube.com/watch?v=-mfL6B0w9UI* .

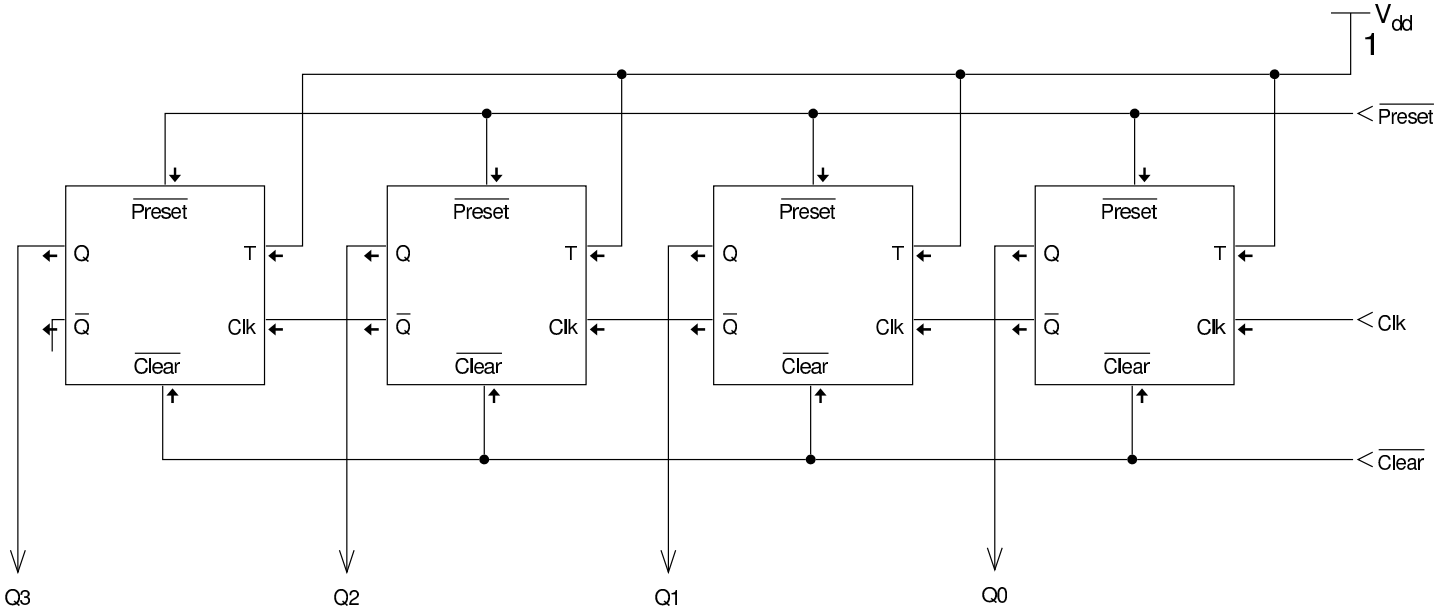
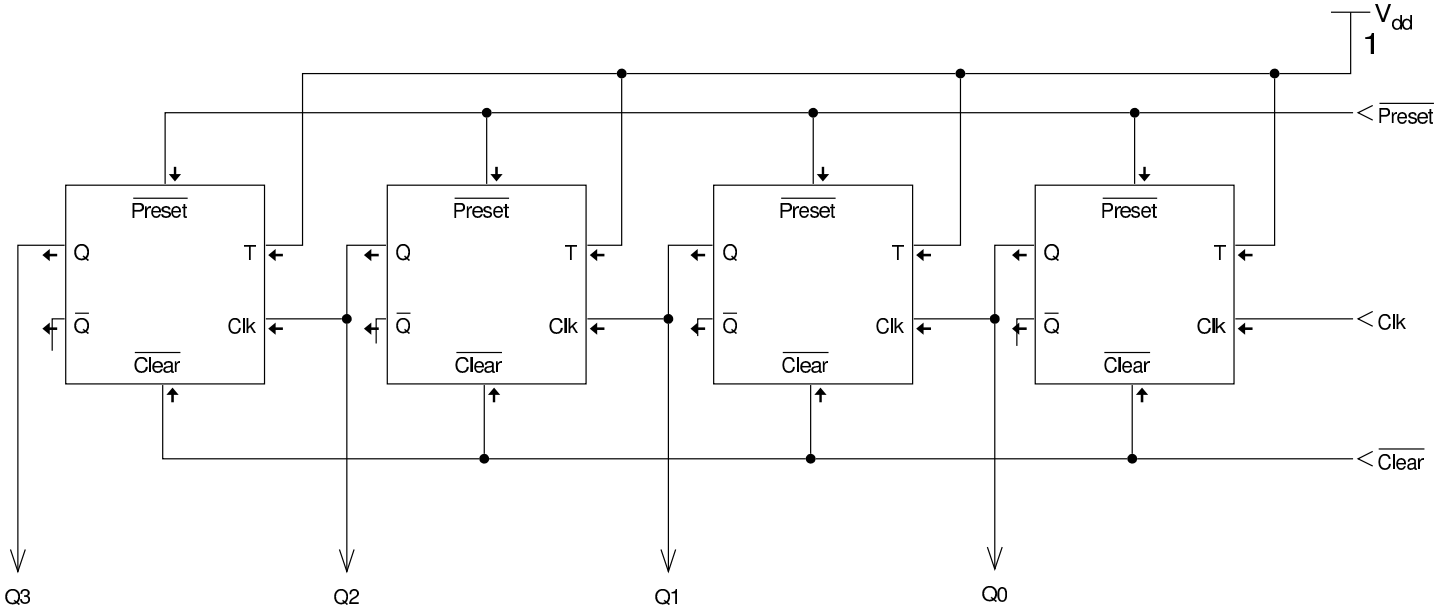


Figure 6.11. 4-bit decreasing asynchronous counter: the next flip-flop is driven by the Q output from the previous one. Video: ogv *http://www.youtube.com/watch?v=fvhina5QEnI* .

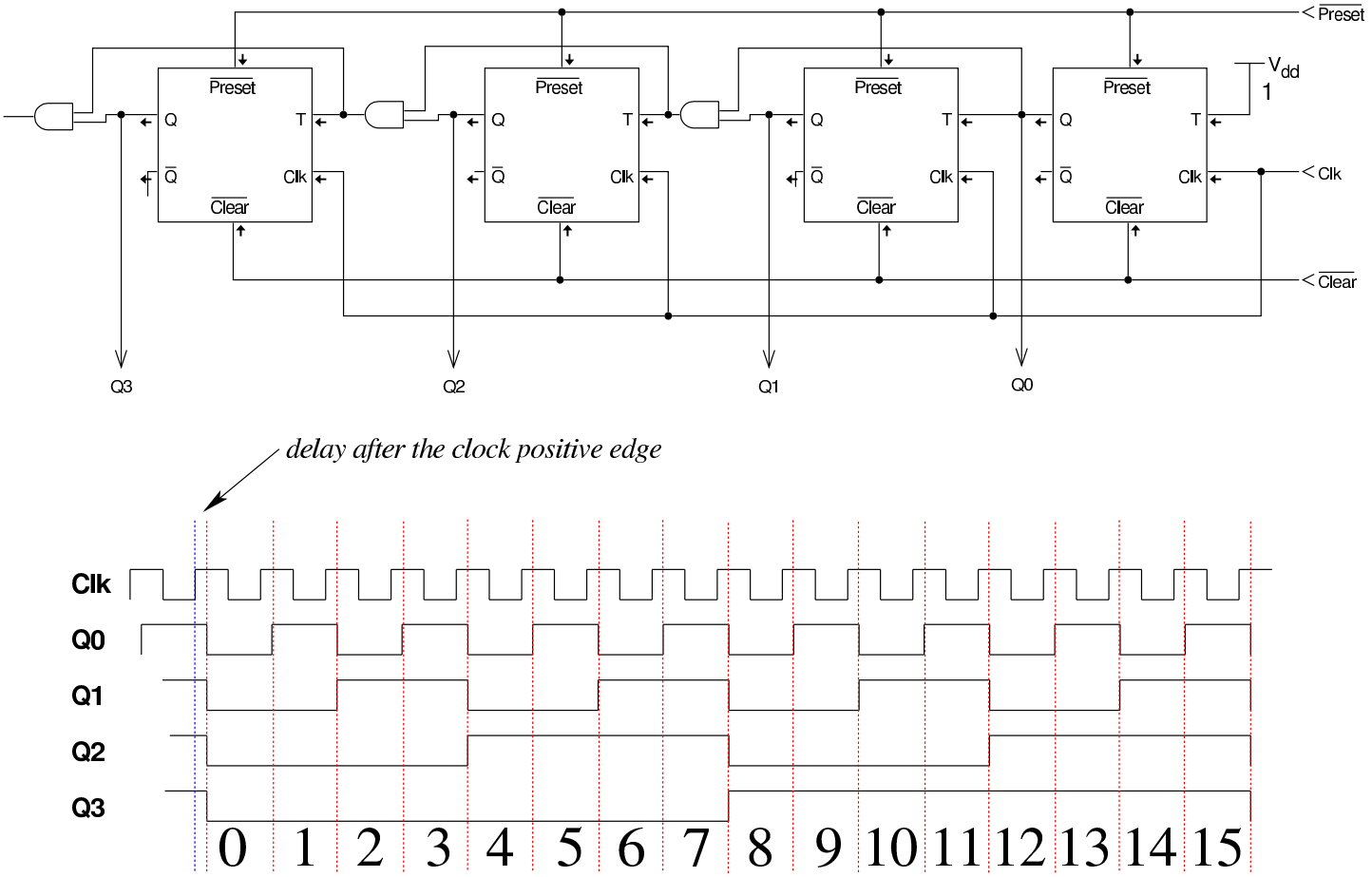


Asynchronous counters have the disadvantage that each binary digit is updated with a little delay, which is added to the next one.

6.4 Synchronous T flip-flop counters

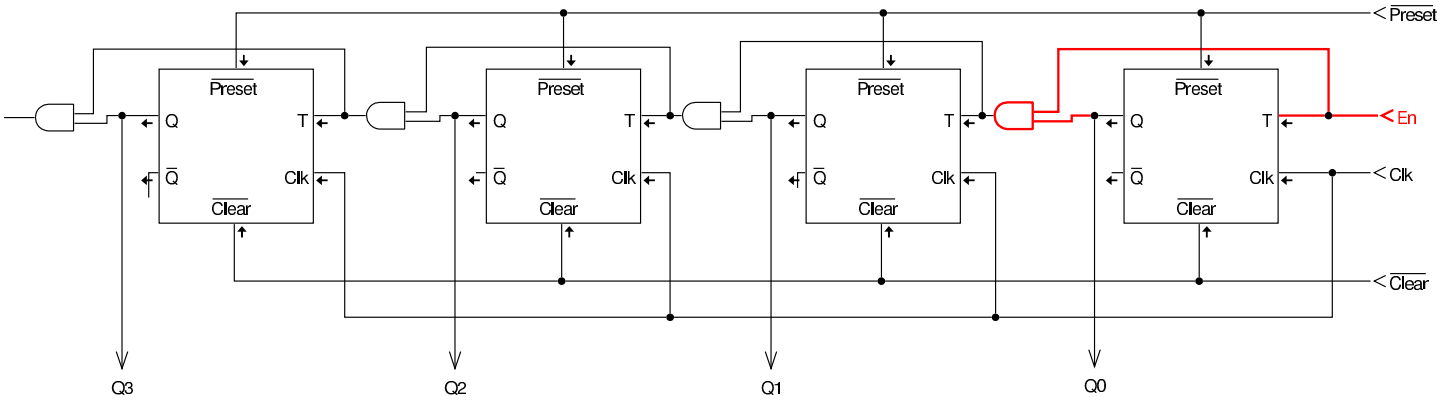
Synchronous counters can be builded with T flip-flop, but in a different way from what is described in the previous section. However, when these counters need to be modified to allow a paralel binary load, D flip-flops should be used instead.

Figure 6.12. 4-bit synchronous counter based on T flip-flops: the outputs are synchronous, but a little later than the clock impulse. Video: ogv *http://www.youtube.com/watch?v=Nh5CzRopNdY* .



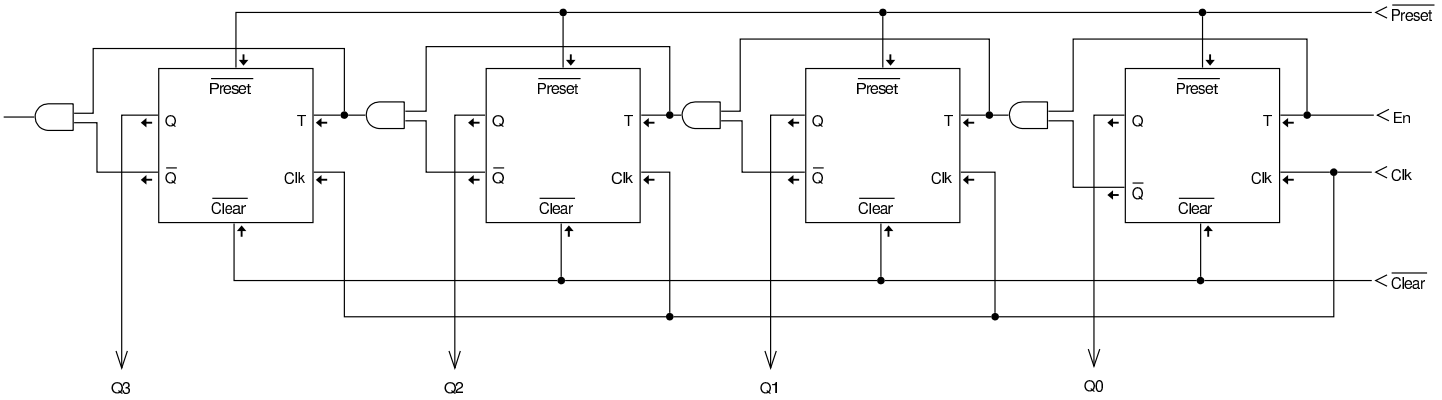
The synchronous counter requires that the ***T*** input of the first flip-flop remains always asserted. However, controlling that input it is possible to pause the counting process, without clearing the current value. Therefore it is possible to add an enable input, controlling the ***T*** input of the first flip-flop, as the following figure shows in red.

Figure 6.13. 4-bit synchronous counter with enable input (***En***).



To obtain a decreasing counter it is enough to invert the outputs of each T flip-flop involved.

Figure 6.14. 4-bit decreasing synchronous counter, with enable input.



6.5 Synchronous counters with D flip-flops

It is generally more convenient to build synchronous counter using D flip-flops, as they can be easily extended to allow a paralel binary load. This section shows only equivalent examples of the previous one (with D flip-flops), but the following section is then related to the paralel load.

Figure 6.15. 4-bit ascending synchronous counter, with enable input, made of D flip-flops.

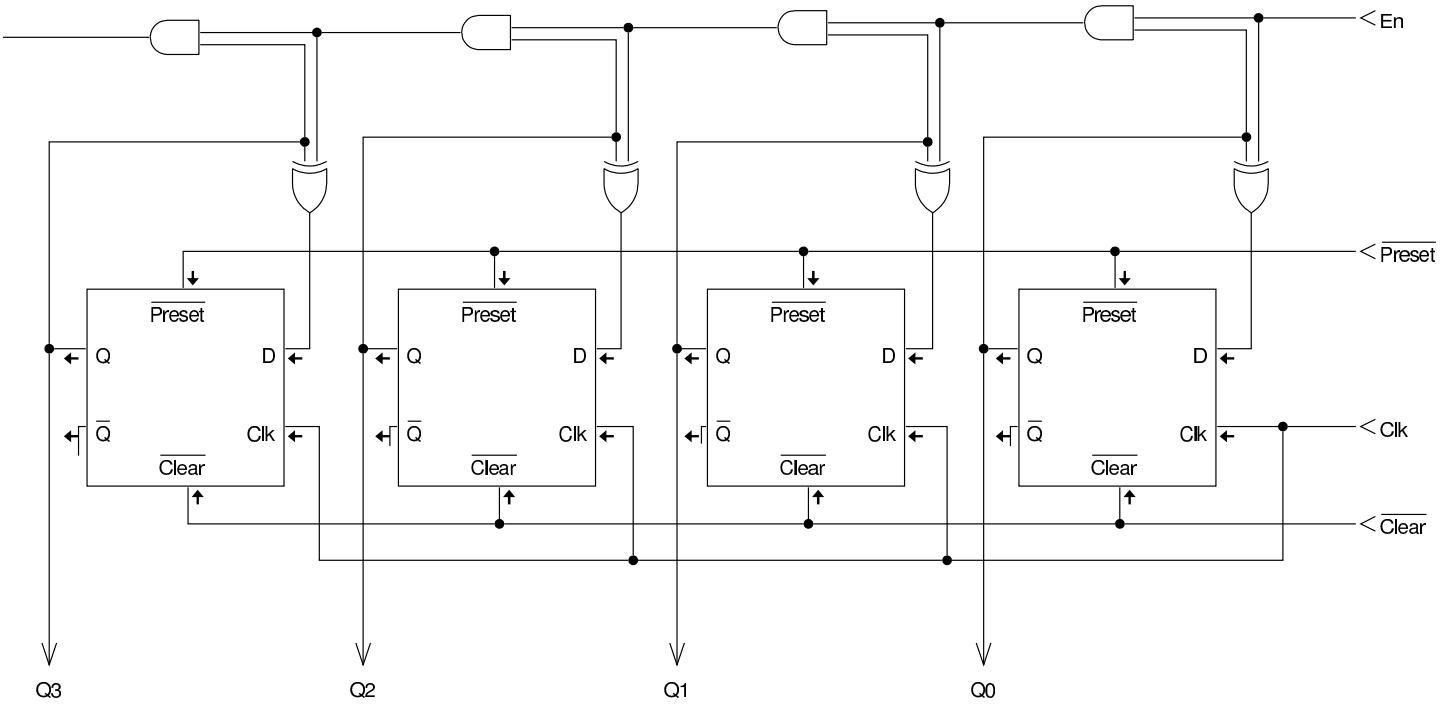


Figure 6.16. 4-bit decreasing synchronous counter, with enable input, made of D flip-flops.

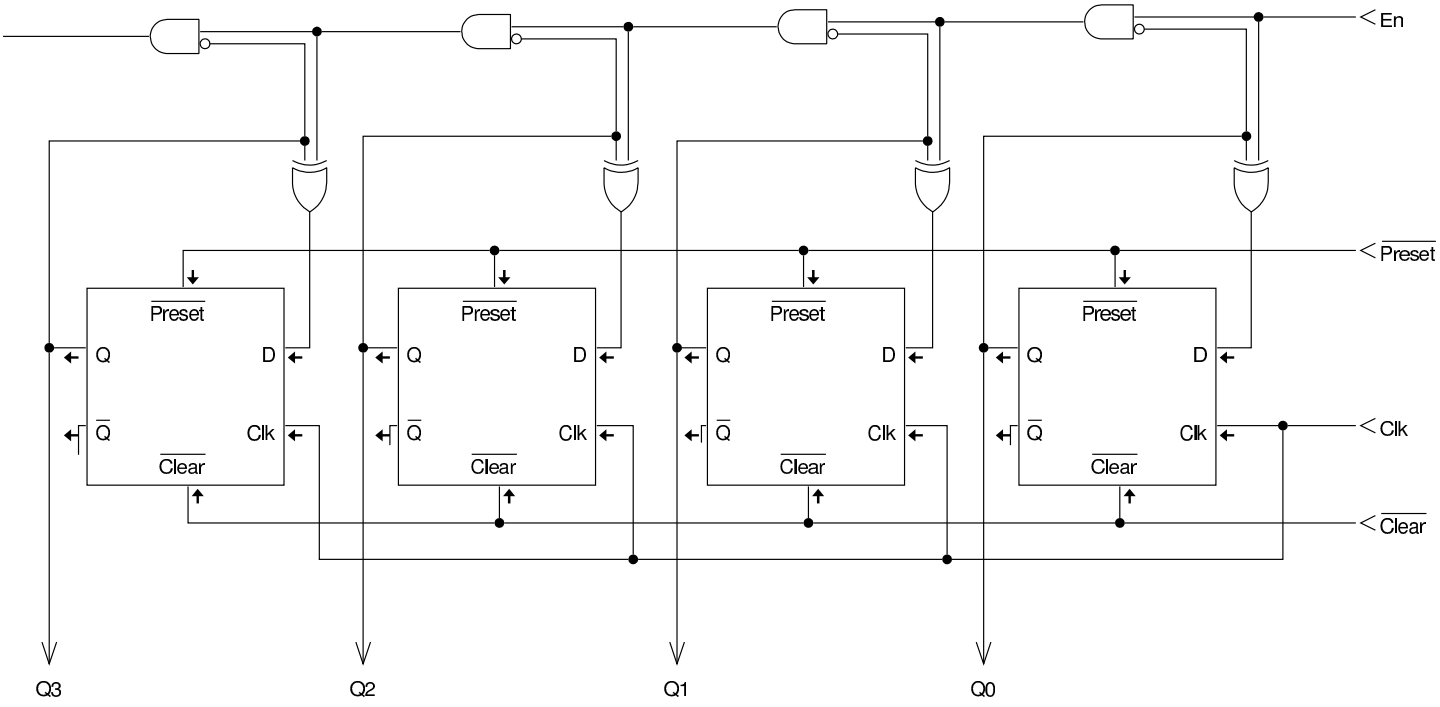
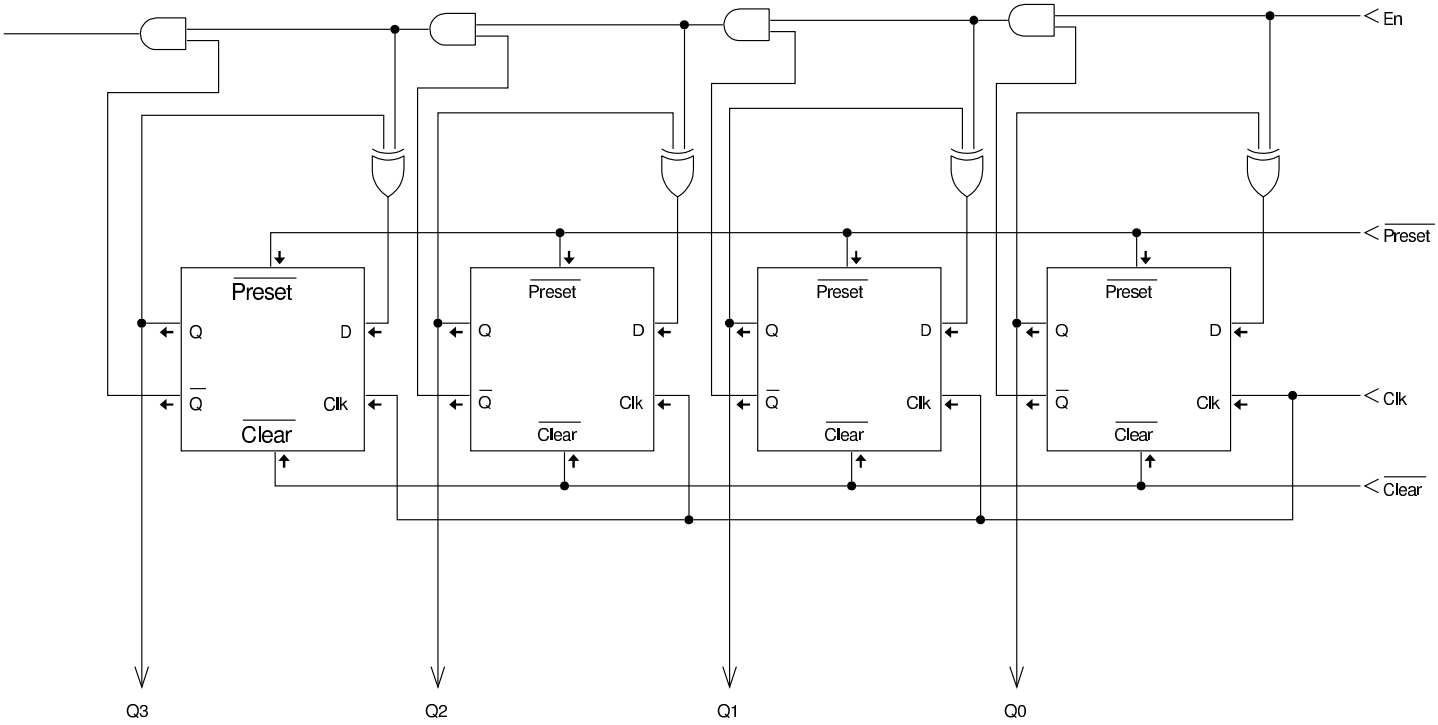


Figure 6.17. Alternative 4-bit decreasing synchronous counter, with enable input, made of D flip-flops.



It is important to understand the logic added to the D flip-flops, which is made of simple half-adders or half-subtractors.

Figure 6.18. The ascending synchronous counter contains a half-adder.

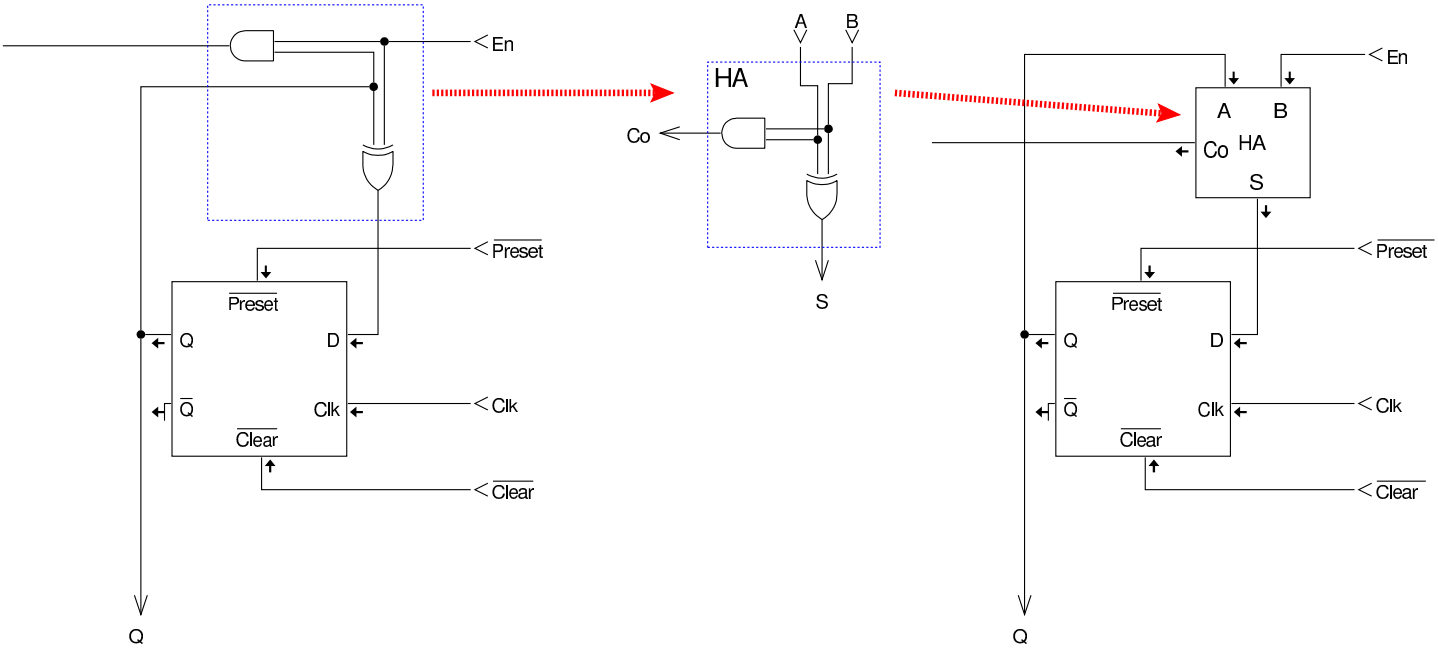
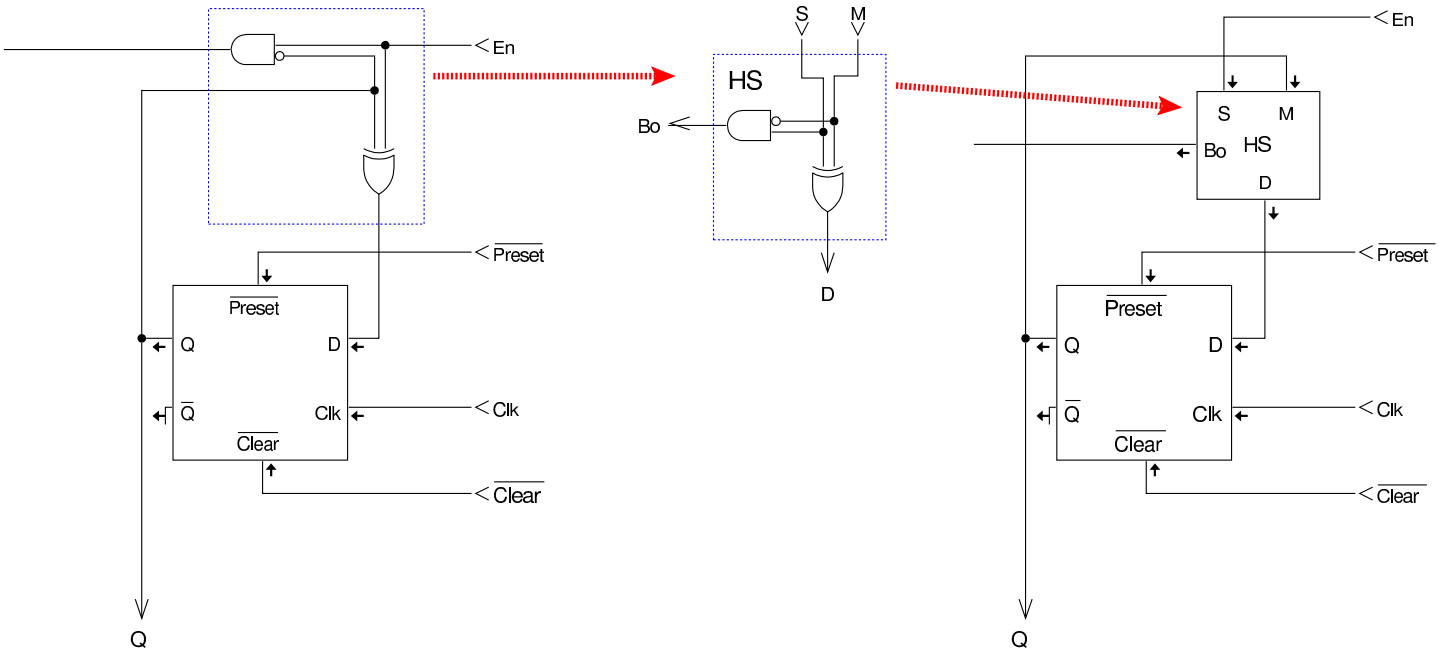
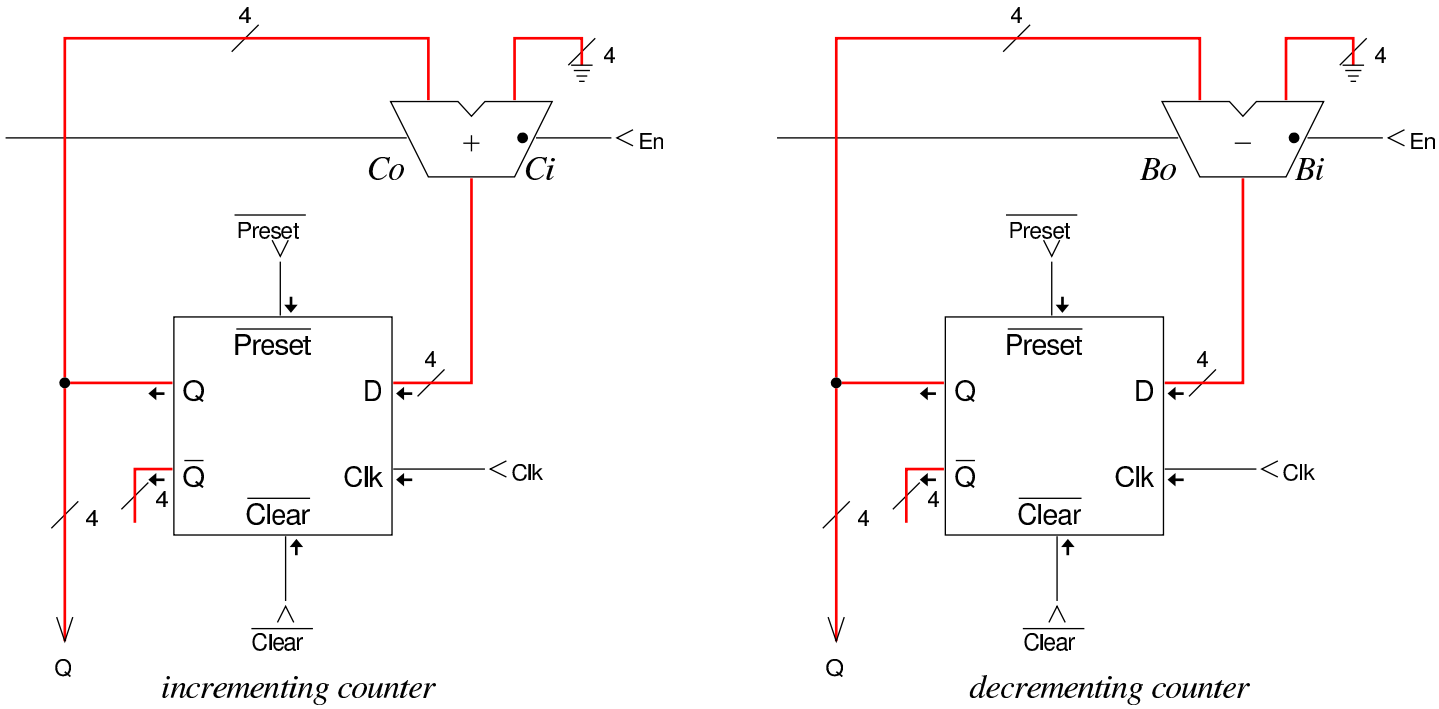


Figure 6.19. The decreasing synchronous counter contains a half-subtractor.



The D flip-flops might be represented in a compact, paralel draw, where the ports ***D***, ***Q*** and $\overline{\boldsymbol{Q}}$ are bit vectors. In that case also the counter can be shown in a more compact way. The following figure shows the use of a full-adder and a full-subtractor, because they allow to follow the concept in an easier and immediate way.

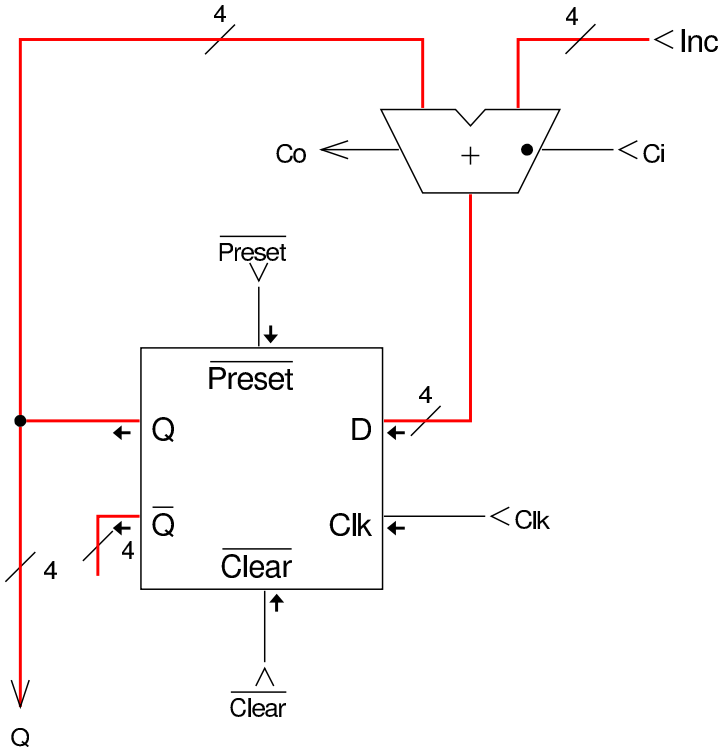
Figure 6.20. Compact representations.



The compact representation allows to understand how to extend and possibly to sim-

plify the logic. The following example replaces the enable input with the value to be added at each clock signal, value that might be either positive or negative, or even zero if no change should be applied.

Figure 6.21. More general counter with variable increment.



6.6 Synchronous counters with paralel load

The counter is complete only if it allows the paralel data load. The following figures show counters based on D flip-flops, allowing the paralel data load.

Figure 6.22. 4-bit synchronous counter with paralel data load: the input ***Load*** allows to load the value received by the inputs $\boldsymbol{D}_{3..0}$ into the flip-flops; otherwise, if the ***Load*** input is negated, the input ***En*** allows to increment or keep unchanged the previous value stored inside the register.

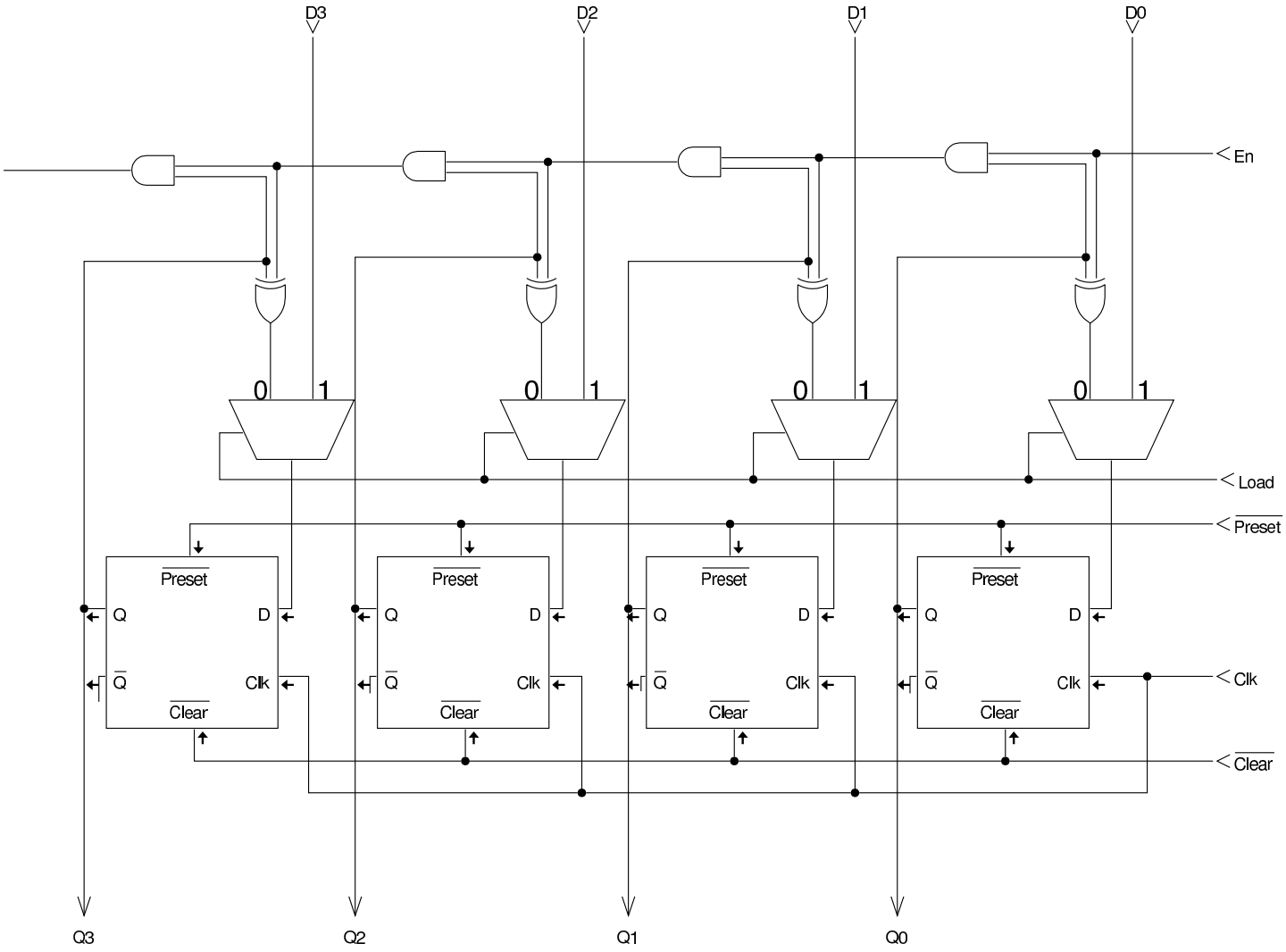


Figure 6.23. 4-bit synchronous decreasing counter with paralel load, similar to the previous figure.

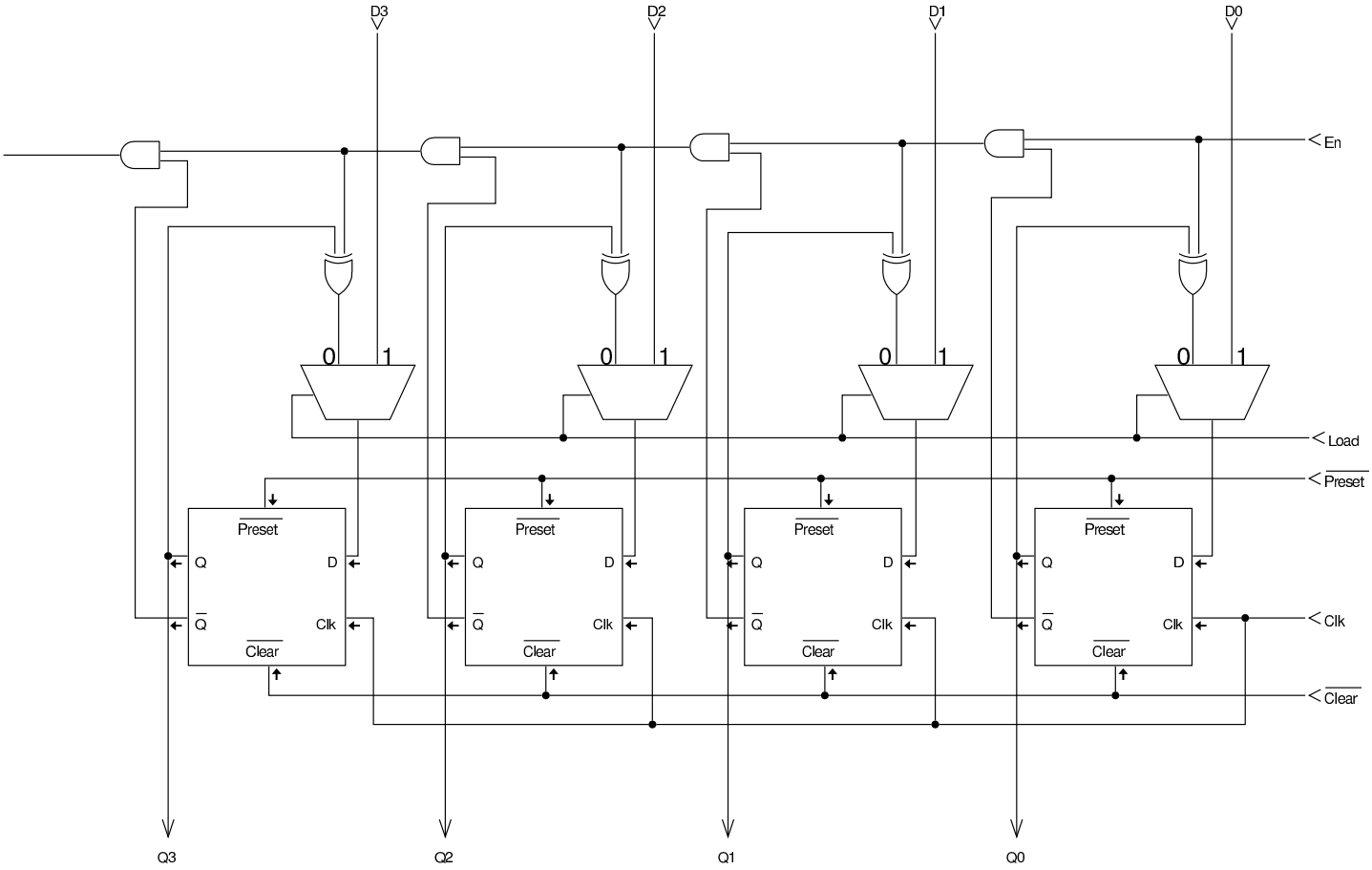


Figure 6.24. 4-bit synchronous increasing and decreasing counter: the input ***f*** allows to select if the counter should increase or decrease.

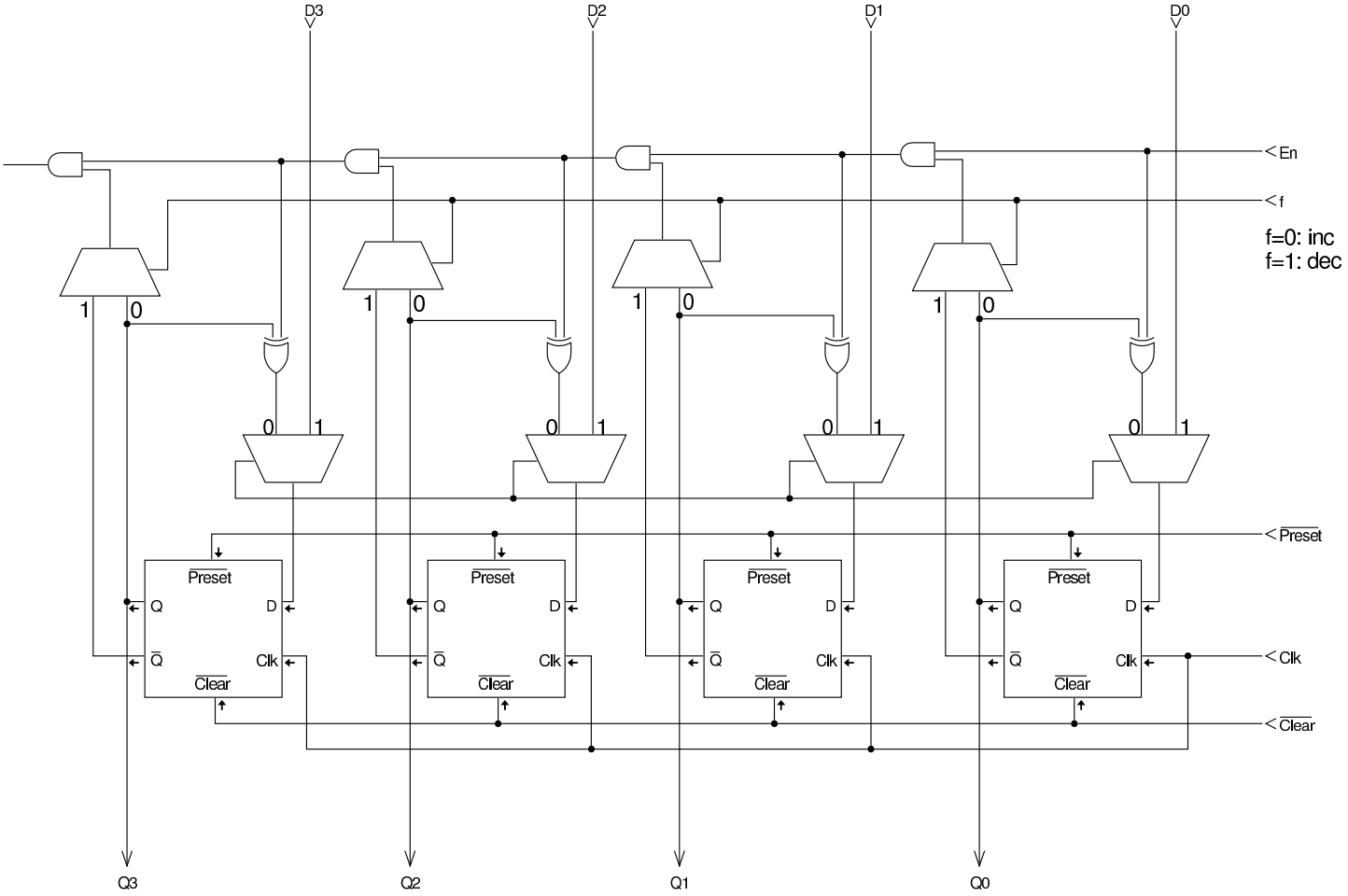
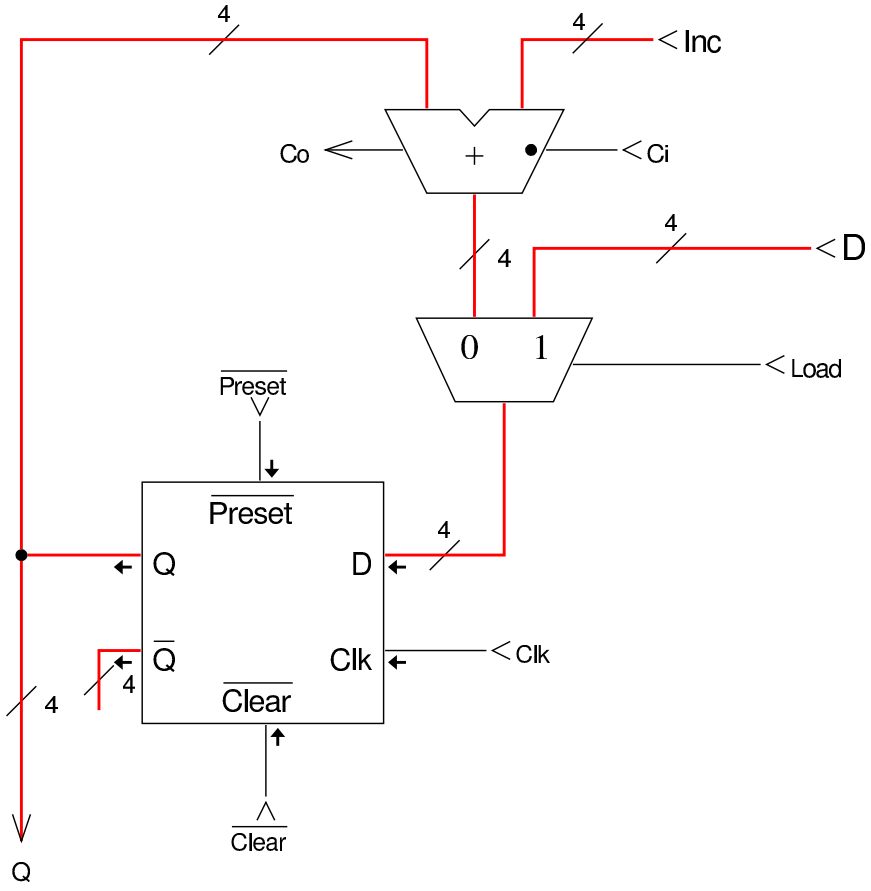


Figure 6.25. 4-bit synchronous counter with paralel load: this is a simpler solution, where the increment value is free and might be positive, negative or zero.



7.1 Tri-state buffer 229
7.2 Hardcoded bus control unit 232
7.3 Microcode 236

bus 229 microcode 236 tri-state buffer 229

Registers and combinational circuits are often grouped together, joining a set of connections (wires). This set of connections is what is known as ***bus***. A bus where different components have the ability to write to and read from is usually a ***data bus***, because it is used to exchange data between the components; however, it should be noticed that only one component at a time should be allowed to write to the bus, whereas for reading there is no limitation.

7.1 Tri-state buffer

A data bus can be implemented with a multiplexer or with ***tri-state buffers***, which are components similar to buffers or inverters (depending on the type), but with an additional input that is used either to enable the output or to isolate it. The following figure shows the tri-state buffer compared to a traditional switch:

0 = isolation
1 = buffer

0 = open circuit
1 = closed circuit

The tri-state buffers, or the multiplexer, are necessary to control the bus writing. The tri-state buffer method is simpler and cheaper than the multiplexer more simple and cheaper, because the multiplexer would require more logic gates and a longer propagation delay.

The following figures show the adaptation of a simple data register, using a tri-state buffer to connect it to a bus; then this kind of register is shown connected with others on a data bus, controlled by a control bus.

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

Figure 7.2. Simple data register adaptation for a data bus.

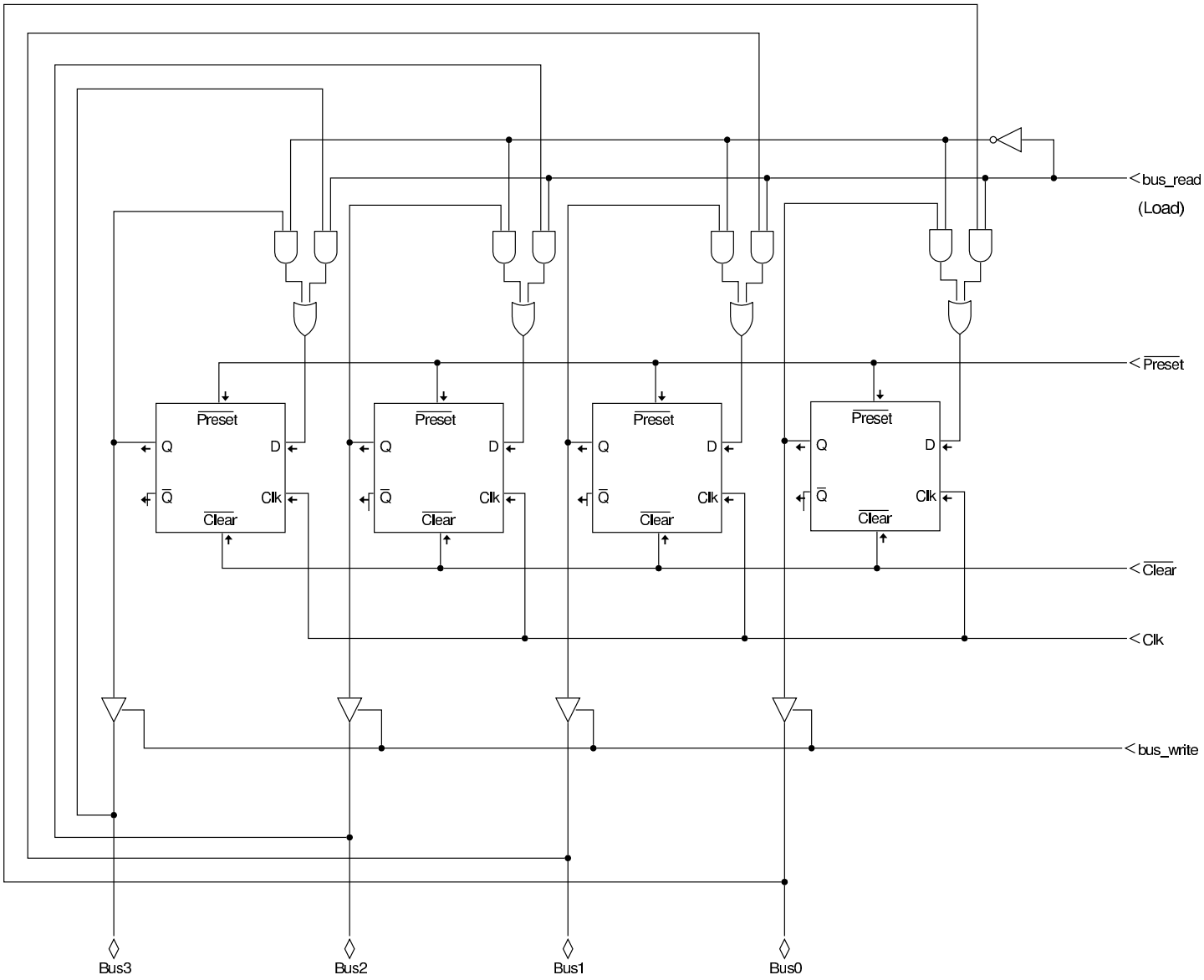
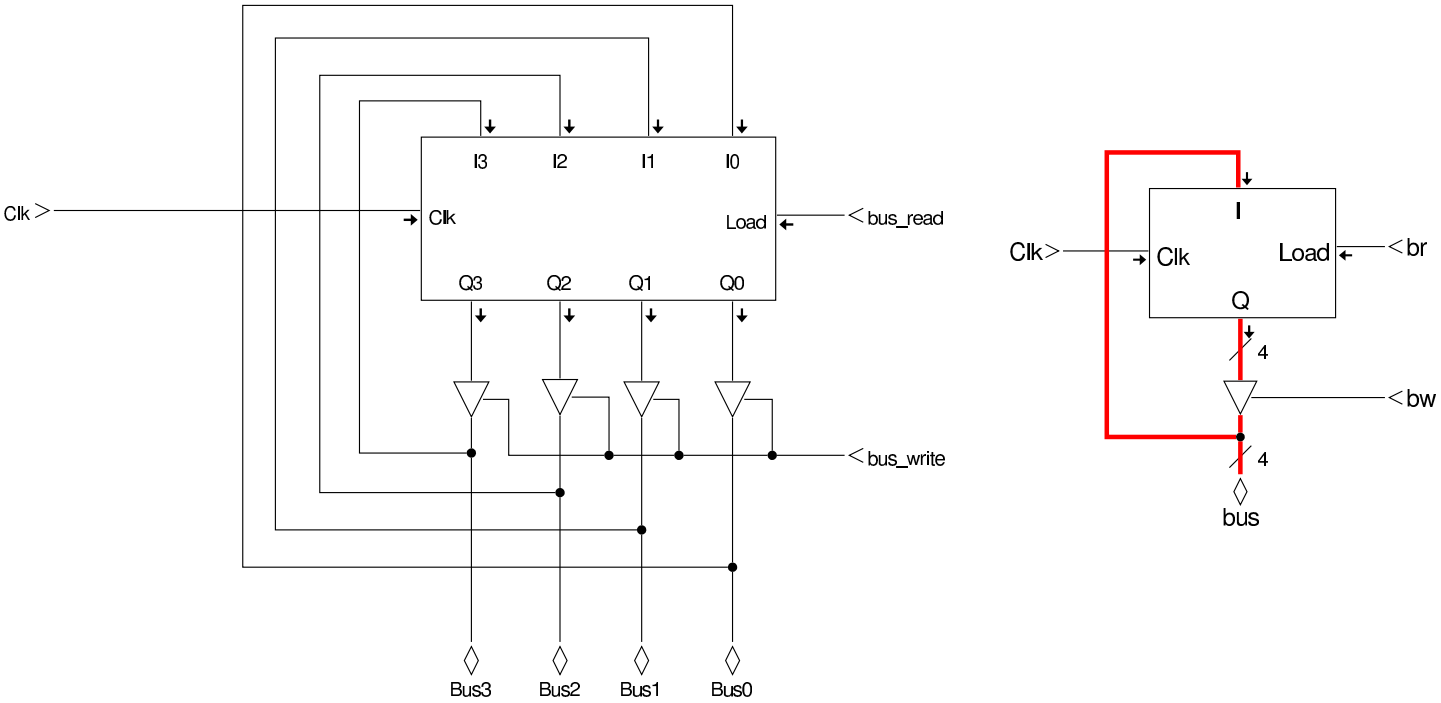
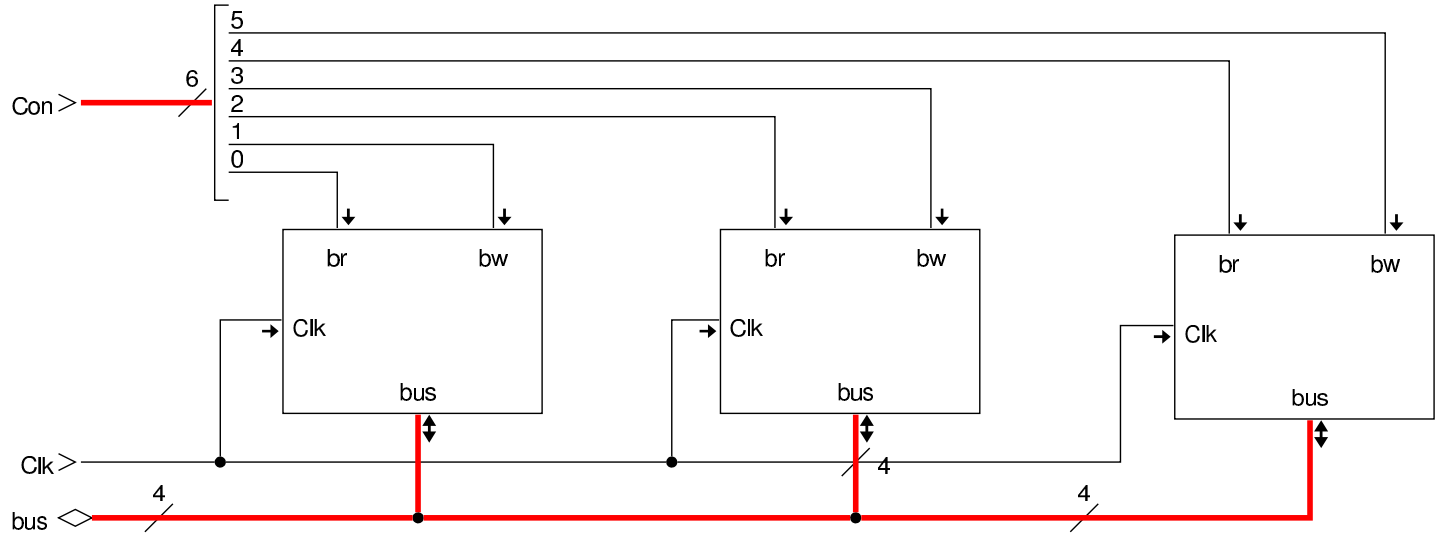


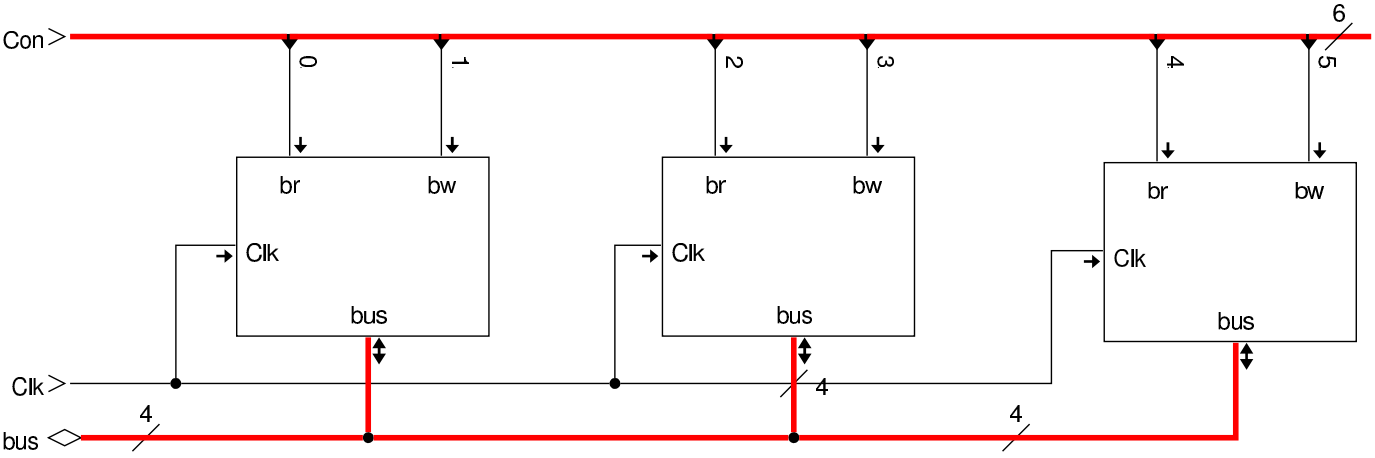
Figure 7.3. Simplified diagrams for the same register adaptation. The input ***bus_read***, or ***br***, is used to load and store the bus value into the register, whereas the input ***bus_write***, or ***bw***, allows the register to write its stored value to the bus.



Here are used the definitions “bus read” and “bus write”, to identify respectively the actions to load a value from the bus and to put a value to the bus; however, some more formal documentation, when referring to registers, prefer the definitions “load” and “enable” instead (because the register loads data or is enabled to write its data).

Figure 7.4. Different components connected to the bus: the control inputs used to enable reading or writing the bus, together with other potential control inputs, are connected to a secondary bus (the control bus). The second draw shows an alternative way to represent the connections to the control bus, picking a wire at a time.

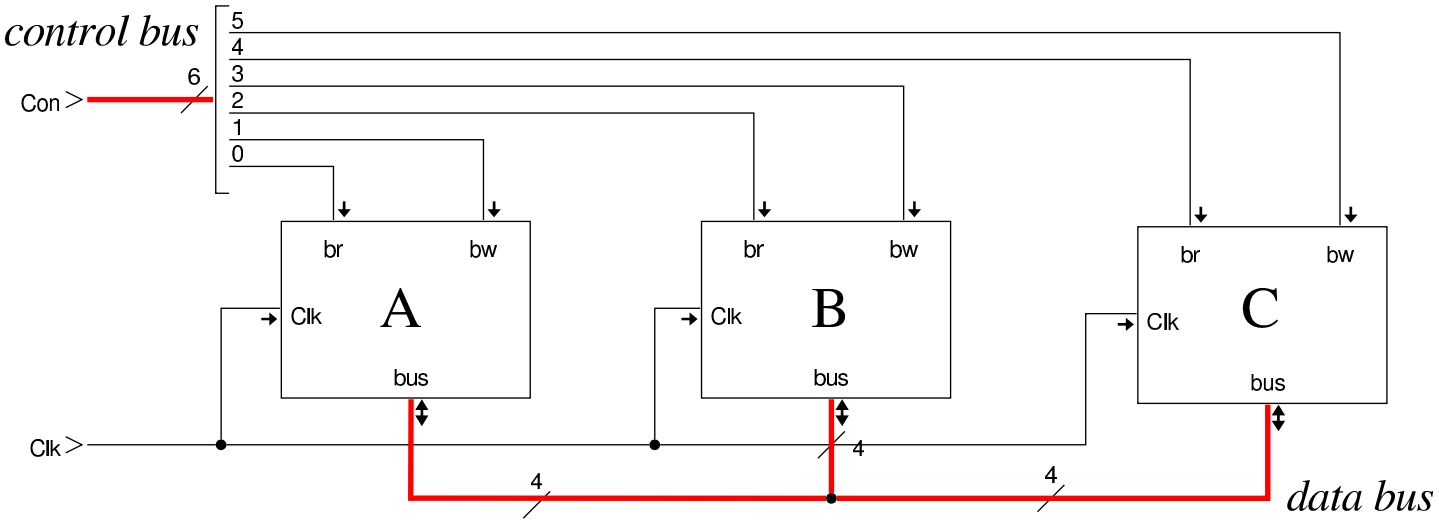




7.2 Hardcoded bus control unit

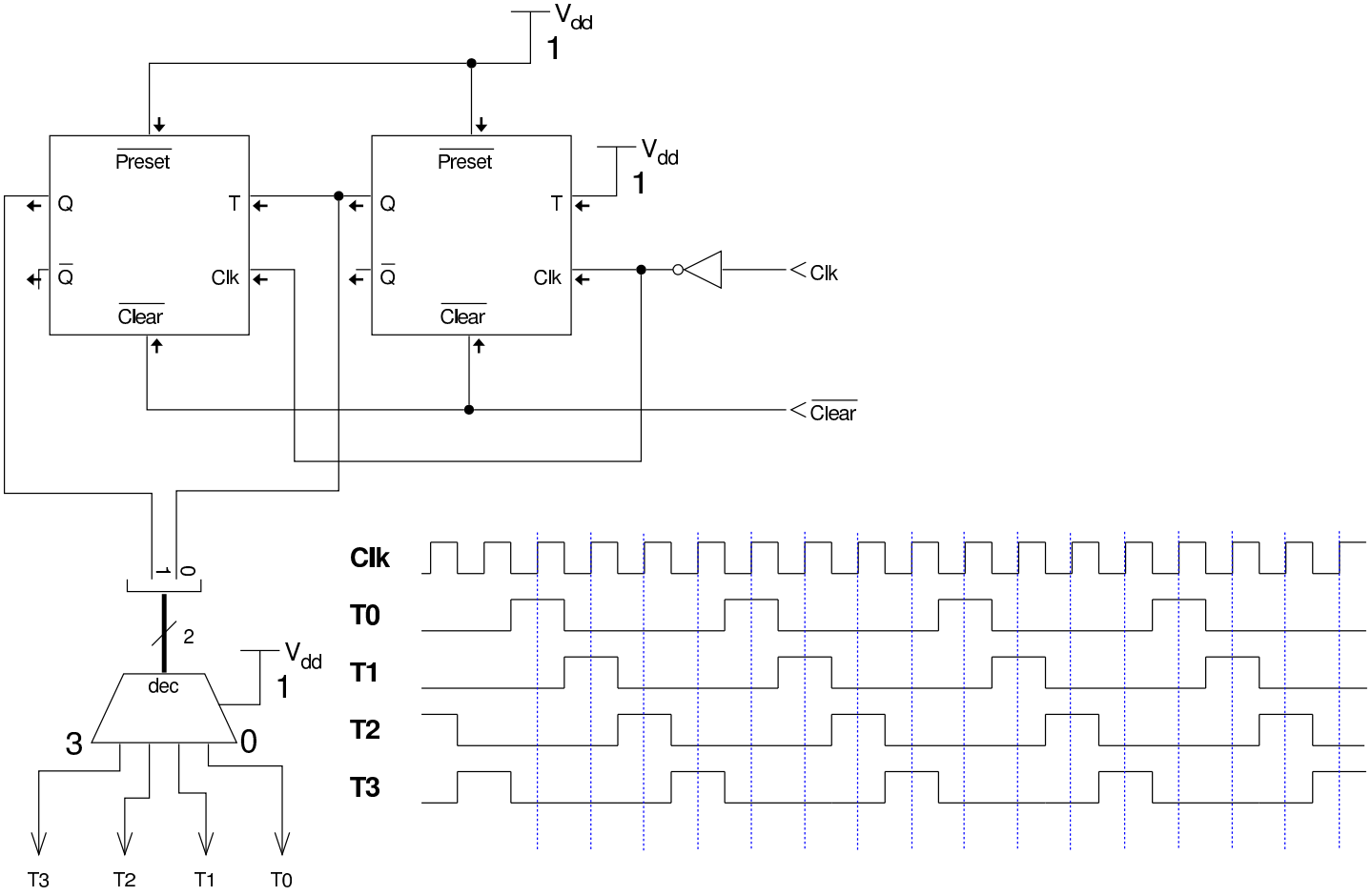
To direct properly the data bus usage it is necessary a ***control unit***, which should be responsible to mark the phases of every process that is to be applied. To understand the problem the following diagram is used (which is already appeared in the previous section), where three components communicate through a data bus, under the direction of a control bus: for these components are considered only data transfer operations.

Figure 7.5. Three components communicating through a data bus.



To mark the operation steps, the control unit needs a counter. There are two main types of control units, the first one is ***hardcoded***, the second is ***microcoded***. The hardcoded control unit requires a shift counter, which can be implemented as the following figure shows. What should be noticed reading the timing diagram is that the output control lines are active (asserted) before the clock impulse used to drive the data bus components: to obtain this result, the clock pulse used by the counter is inverted.

Figure 7.6. Shift counter used by the control unit: the clock signal is inverted, so that the output lines T_n are already active when clock positive margin reaches the data bus components. Video: ogv *http://www.youtube.com/watch?v=vXco4E4SNTo* .



The hardcoded control unit should put together a function decoder with the shift register outputs. For example, the function f_1 of the diagram should do the following: copy A to B, then copy B to C: the decoder activates the second line (labeled $f1$), then, with some AND gates the outputs Aw and Br are activated when the shift counter has the T_0 output active, then, the same way, the outputs Bw and Cr are activated when the shift counter has the T_1 output active. The outputs Aw, Br, Bw and Cr, activated at the proper time, following the shift counter direction, would enable the registers read-write, allowing the programmed data transfer. The control unit implementation that appears in the following figure is extremely simplified and the T_2 time is not even used; however, the time T_3 is used instead to stop the clock signal related to the control unit itself, which should be cleared to be able to accept a new command from the input f.

Figure 7.7. Control unit: only four simple functions are available, for which two times are sufficient; therefore the time T_2 is unused and the time T_3 locks the control unit until the ***Run*** input is cleared and reactivated. Video: ogv *http://www.youtube.com/watch?v=r-DZgjy-ya0* .

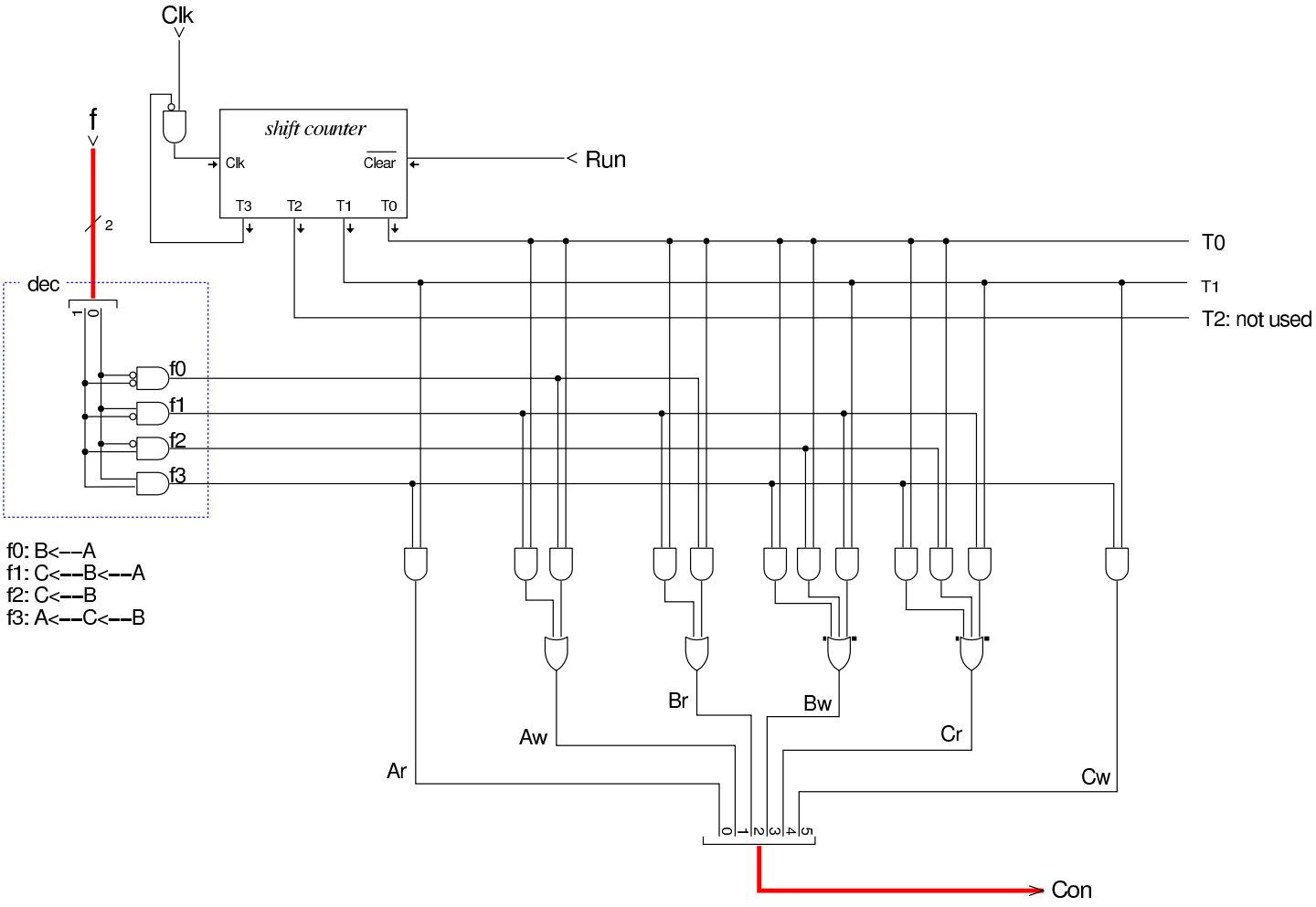
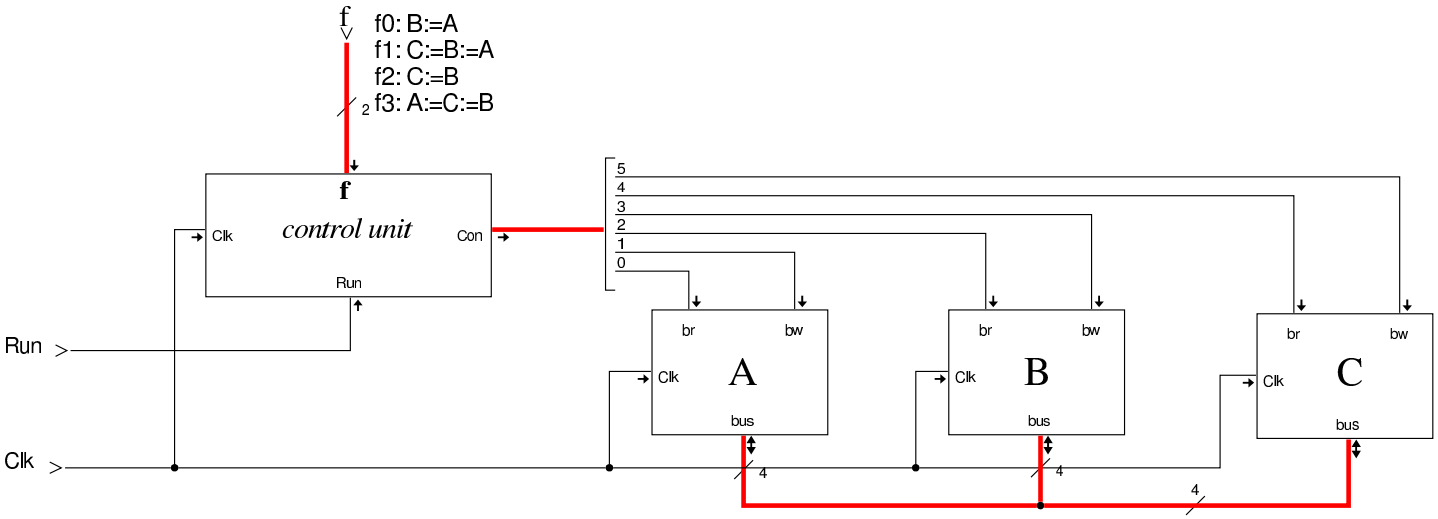


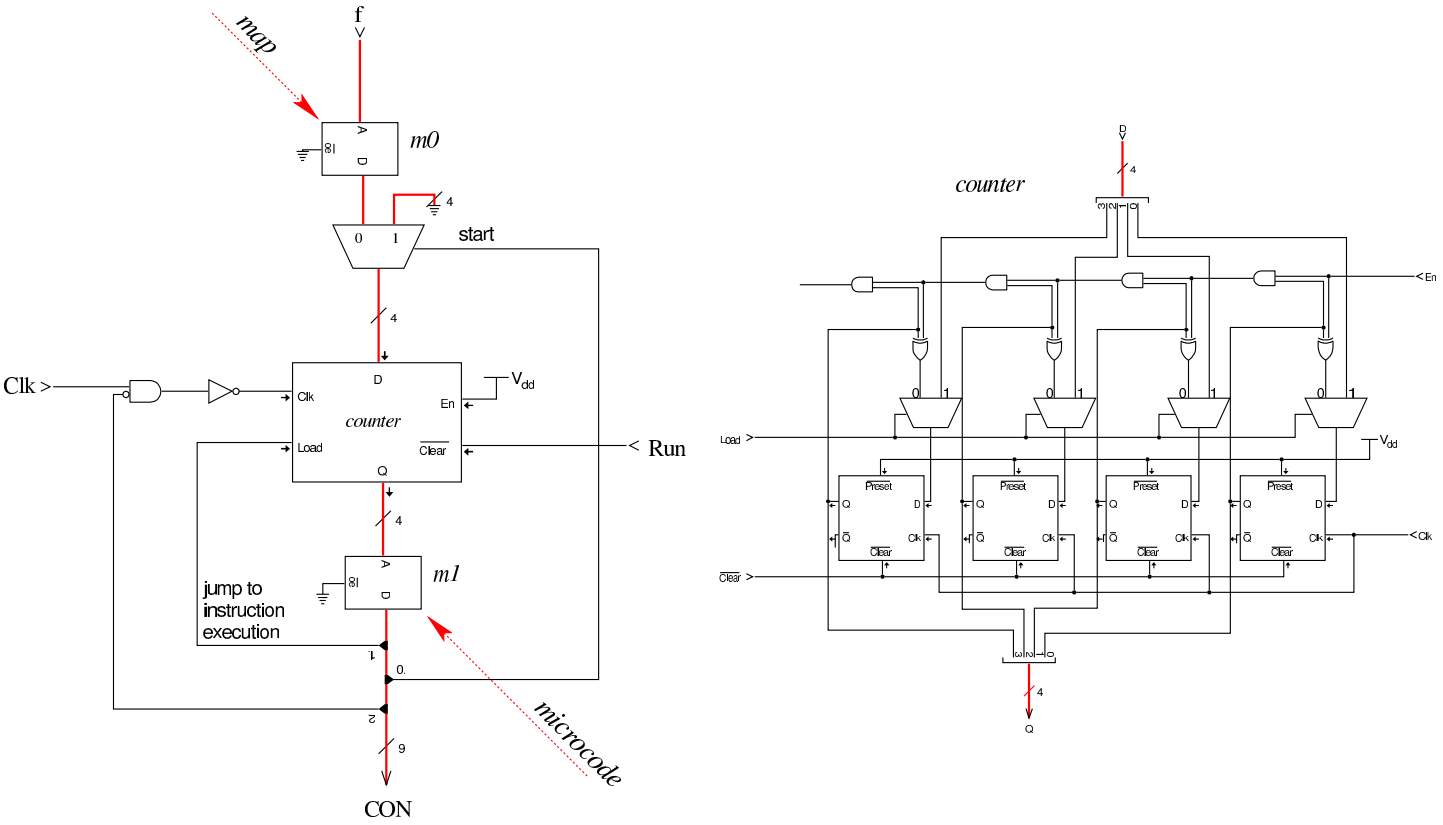
Figure 7.8. Control unit connected to the components that should drive. Video: ogv *http://www.youtube.com/watch?v=kpET2kEcUIo* .



7.3 Microcode

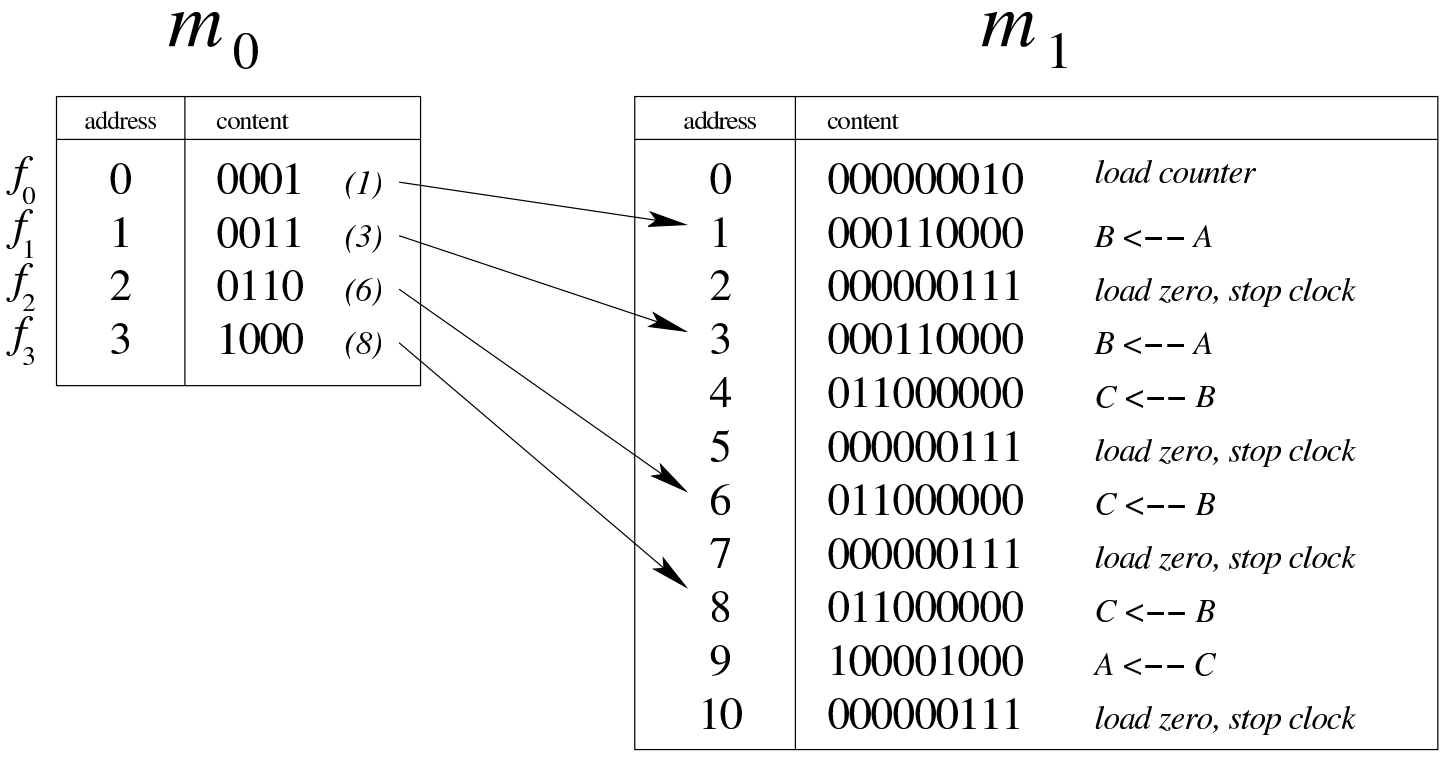
The implementation of a hardcoded control unit is a complex work. For that reason is usually preferable a ROM memory (read only), divided into two parts: the ***map*** and the ***microcode***. The two parts of the memory should be see as two tables: the first one translates the desired function into an address to the second table, where the code for the related procedure starts. Please see the following figure, where on the left there is the control unit diagram and on the right there is the counter implementation.

Figure 7.9. Control unit implemented with ROM memories: on the left appears the control unit block diagram and on the right there is the detailed four-bit counter (the counter can be incremented or can load a new value). Video: ogv *http://www.youtube.com/watch?v=mBGhNP3Uujs* .



The two ROM memories are named, respectively, ***m0*** and ***m1***. In the following figure is shown the content of these memories, using tables, associating the memory address input (***A***) with the corresponding value that is returned from the memory data output (***D***). The ***m0*** memory of these examples uses only two bits for the addresses, which correspond to the function requested to the control unit that is known as ***operation code*** or ***opcode***, whereas the data output has a 4-bit range, which in turn is the address for the ***m1*** memory (the microcode).

Figure 7.10. Memory contents and links between them.



The output from the ***m0*** memory is assigned to a counter; the counter output is used to access to the ***m1*** memory; the ***m1*** memory output is the ***microcode word*** used to control the bus and the same control unit itself. To allow a better understanding of the process, the various steps are listed below, assuming that the function $\boldsymbol{f}_1$ (opcode) has been requested.

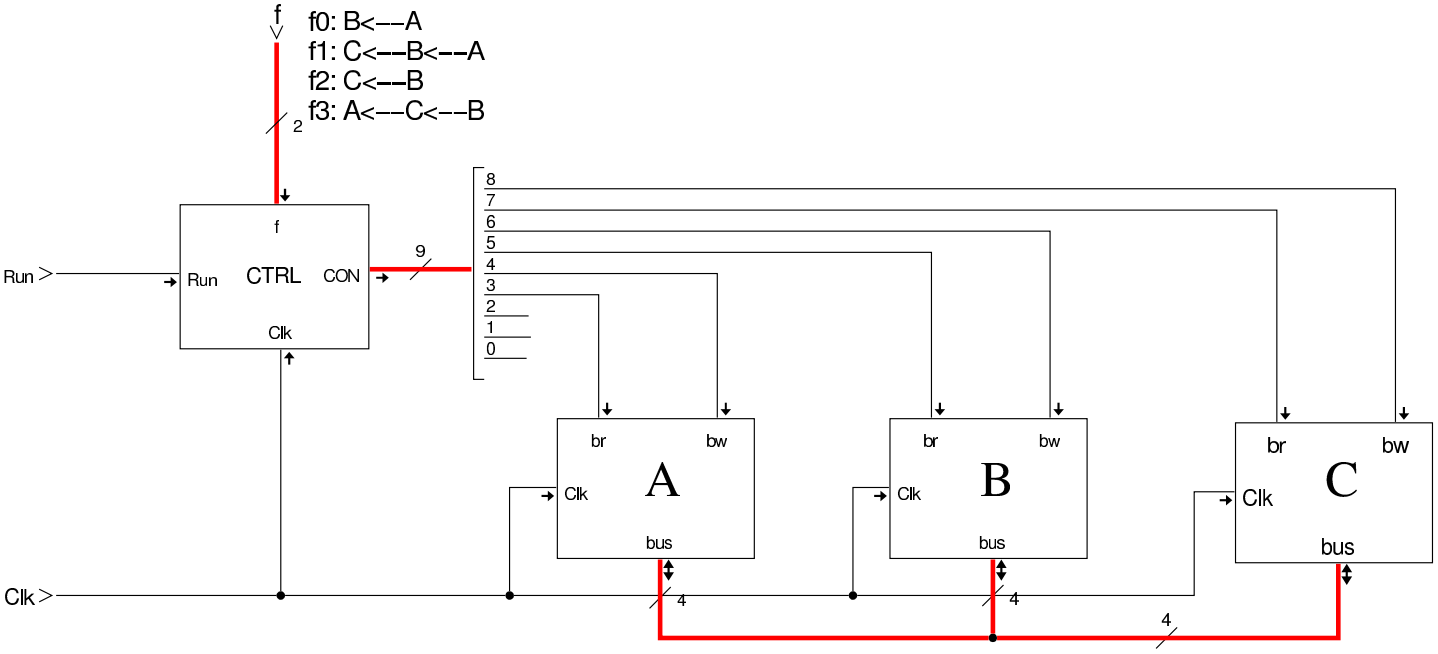
1. When the system is started, the ***Run*** input is equal to 0 (zero), which involves the counter reset to zero. As the counter starts from zero, from the ***m1*** memory is selected the value 000000010_2, which is the initial microcode word. The active bit of the microcode word is used to request the counter to load the value that comes from the memory ***m0*** (0011_2), which produces that value corresponding to the address that represent the input function ($\boldsymbol{f}_1 = 01_2$).
2. Activating the ***Run*** input, at the clock **negative edge** (the global clock is inverted for the control unit) the counter loads the address from the ***m0*** memory (0011_2) and offers the same value to the ***m1*** memory as an address. The ***m1*** memory produces the value 000110000_2 as the new microcode word, which corresponds to the request to copy the register ***A*** to ***B***.
3. At the next clock negative edge the counter is incremented, selecting from the ***m1*** memory the address 0100_2 which gives the value 011000000_2, which in turn

corresponds to the request to copy the ***B*** content into ***C***.

4. At the next clock negative edge the counter is incremented again, selecting from ***m1*** the value corresponding to the address 0101_2, which is 000000111_2, which in turn corresponds to the request to clear the counter, loading the fixed value 0000_2, and to suspend the clock signal received by the control unit.

At this point, the execution cycle of a function is finished, but to start another cycle after another function is selected, it is necessary to disable and enable again the input ***Run***.

Figure 7.11. Connection between the control unit and the other data bus components: the first three lines from the control bus are used inside the control unit itself and are not useful to drive the data bus modules.



The ROM memories that are used in a control unit can be coded with the help of tools that allow a symbolic notation. For example, TKGate has a special compiler that produces map and microcode image files, starting from a single source file. Please notice that the function requested to the control unit are defined as labels inside the opcode map memory. The following source listing is compatible with the examples of this section.

Listing 7.12. Opcode and microcode following the TKGate 2 syntax.

```
// MEMORIES:
map       bank[1:0]  m0;
microcode bank[8:0]  m1;
//-----------------------------------------------------
// MICROCODE WORD FIELDS:
field ctrl_start[0];     // set the address to 0
field ctrl_load[1];      // load the counter
field stop[2];           // stop the clock
field a_br[3];           // A <-- bus
field a_bw[4];           // A --> bus
field b_br[5];           // B <-- bus
field b_bw[6];           // B --> bus
field c_br[7];           // C <-- bus
field c_bw[8];           // C --> bus
//-----------------------------------------------------
// FUNCTIONS = OPCODES:
op f0 {
  map f0: 0;
  +0[7:0]=0;
};
op f1 {
  map f1: 1;
  +0[7:0]=1;
};
op f2 {
  map f2: 2;
  +0[7:0]=2;
};
op f3 {
  map f3: 3;
  +0[7:0]=3;
};
//-----------------------------------------------------
```

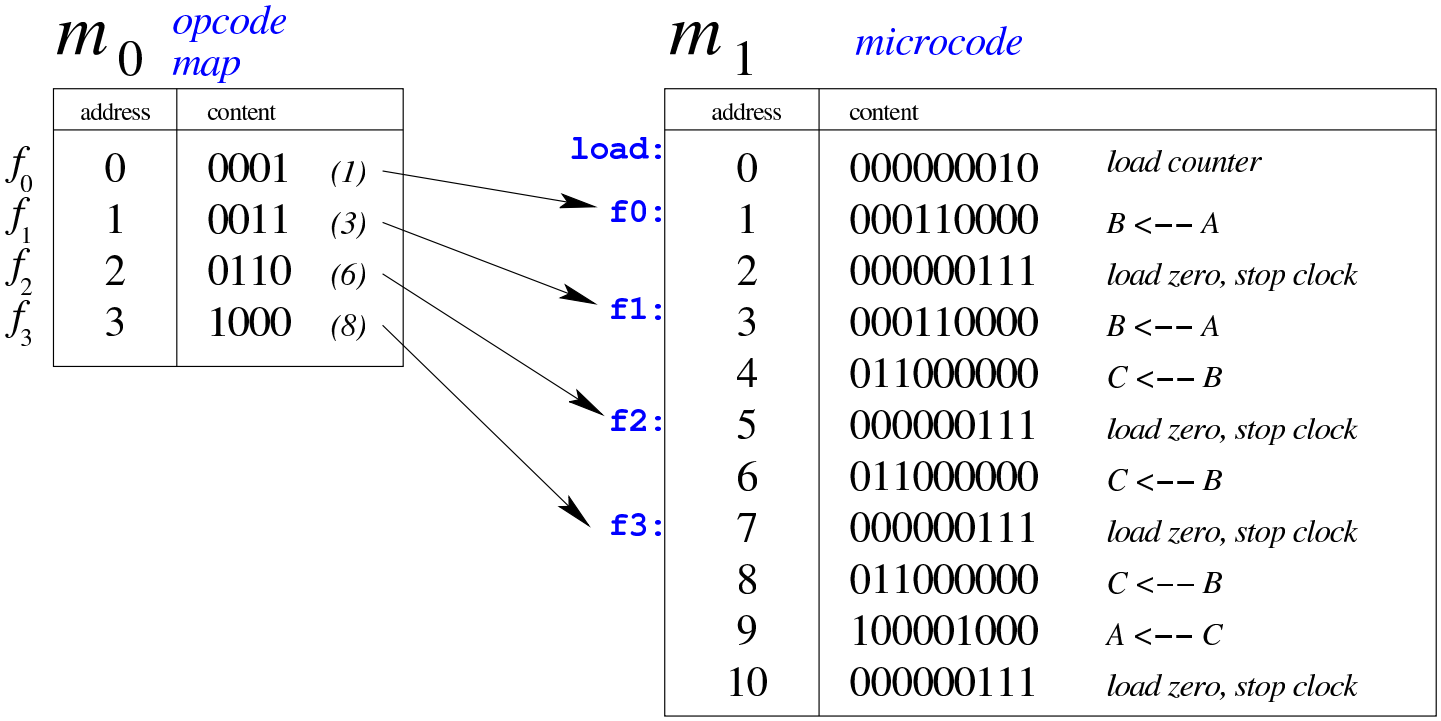
```
// MICROCODE:
begin microcode @ 0
load:
    ctrl_load;                      // CNT <-- TBL[f]

f0:
    b_br a_bw;                      // B <-- A
    ctrl_start ctrl_load stop;      // CNT <-- 0

f1:
    b_br a_bw;                      // B <-- A
    c_br b_bw;                      // C <-- B
    ctrl_start ctrl_load stop;      // CNT <-- 0

f2:
    c_br b_bw;                      // C <-- B
    ctrl_start ctrl_load stop;      // CNT <-- 0

f3:
    c_br b_bw;                      // C <-- B
    a_br c_bw;                      // A <-- C
    ctrl_start ctrl_load stop;      // CNT <-- 0

end
```

The map memory content is organized in opcodes: the operation code is the address in which is located inside the map memory; for example, the function f_2 has the opcode 2 (10_2):

```
op f2 {
  map f2: 2;
  +0[7:0]=2;
};
```

The figure that connects the ***m0*** and ***m1*** memories appears again below, with the addition of other details, to conclude the topic.

Figure 7.14. Map (opcodes) and microcode.



Data distributors or demultiplexers 244
74138 3-to-8 data distributor or demultiplexer 244
74139 dual 2-to-4 data distributor or demultiplexer 247
74154 4-to-16 data distributor or demultiplexer 250
74238 3-to-8 data distributor or demultiplexer 254
Data selectors or multiplexers 257
74150 1-of-16 data selector or multiplexer 257
74151 1-of-8 data selector or multiplexer 262
74153 dual 1-of-4 data selector or multiplexer 265
74157 quad 1-of-2 data selector or multiplexer 268
Special circuits 271
74148 8-to-3 priority encoder 271
Arithmetic units 275
7483 4-bit binary full-adder 275
7485 4-bit magnitude comparator 278
74181 ‘a’ ALU 281
74181 ‘b’ ALU 287
74182 carry lookahead generator 291
74381 4-bit ALU 293
74382 4-bit ALU 297

74138 244 74139 247 74148 271 74150 257 74151 262 74153 265 74154 250 74157 268 74181 281 287 74182 291 74238 254 74381 293 74382 297 7483 275 7485 278

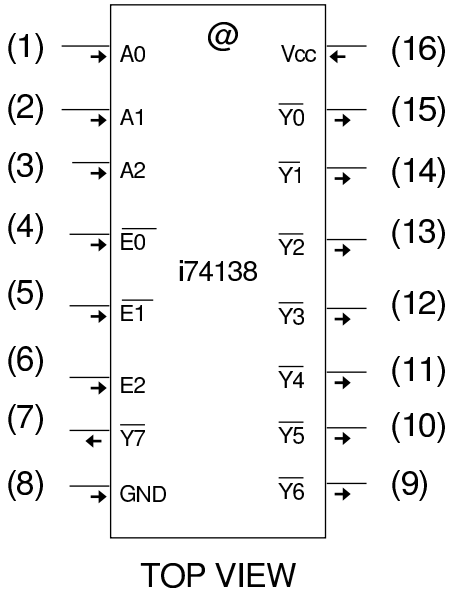
“from the ground up” draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

Data distributors or demultiplexers

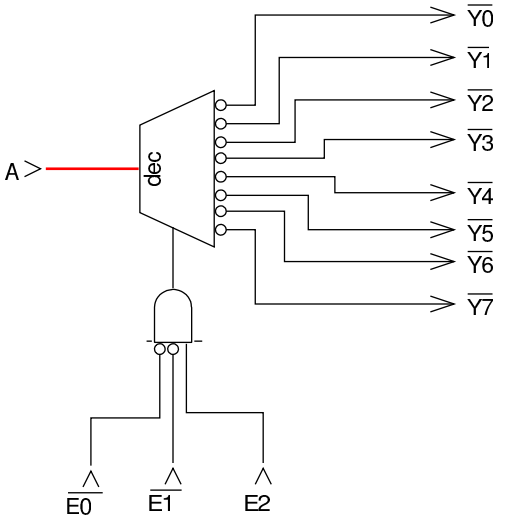
74138 3-to-8 data distributor or demultiplexer

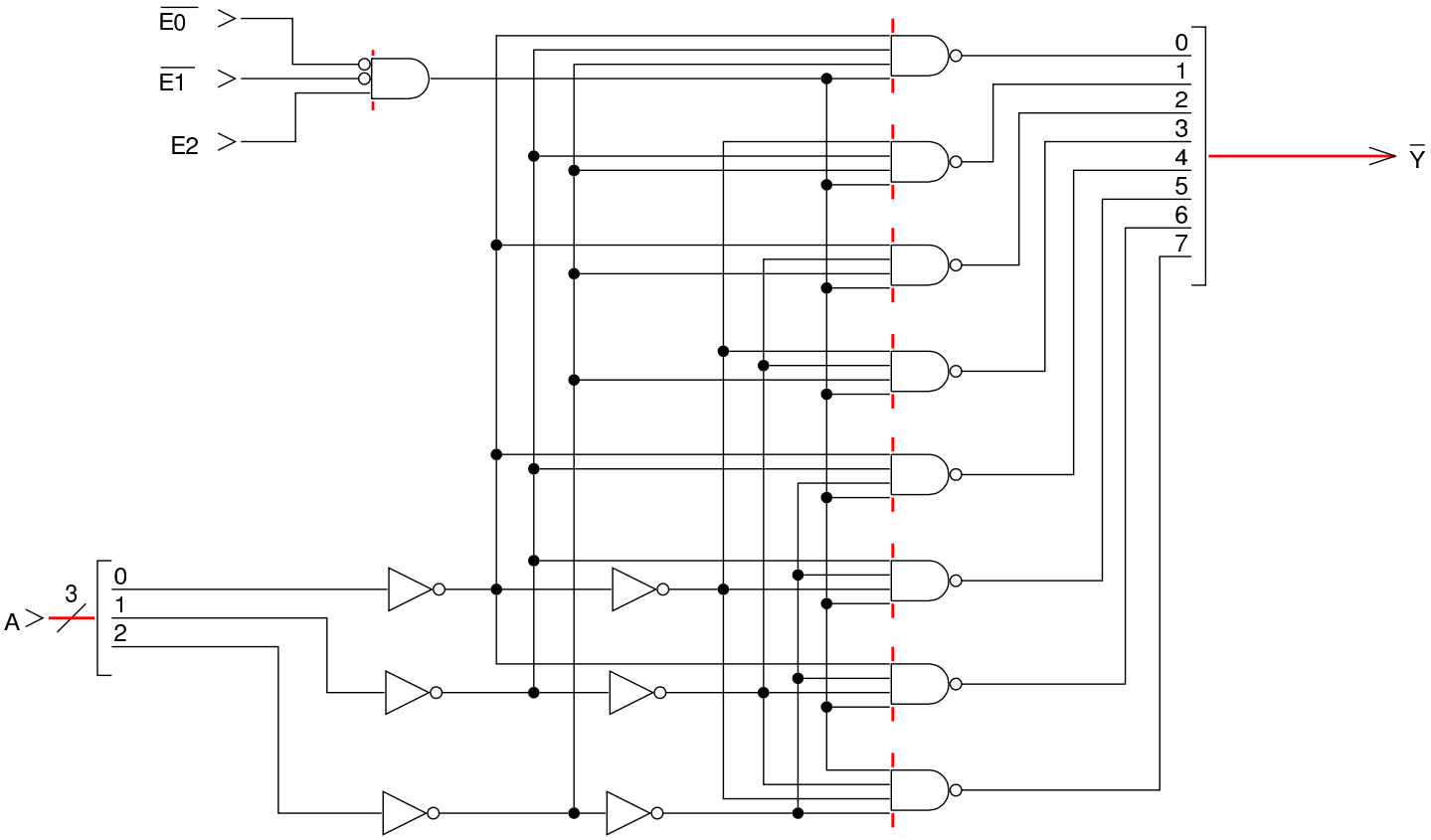
74138 truth table.

E_2	$\overline{E_1}$	$\overline{E_0}$	A_2	A_1	A_0	$\overline{Y_7}$	$\overline{Y_6}$	$\overline{Y_5}$	$\overline{Y_4}$	$\overline{Y_3}$	$\overline{Y_2}$	$\overline{Y_1}$	$\overline{Y_0}$
X	X	1	X	X	X	1	1	1	1	1	1	1	1
X	1	0	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1



TOP VIEW





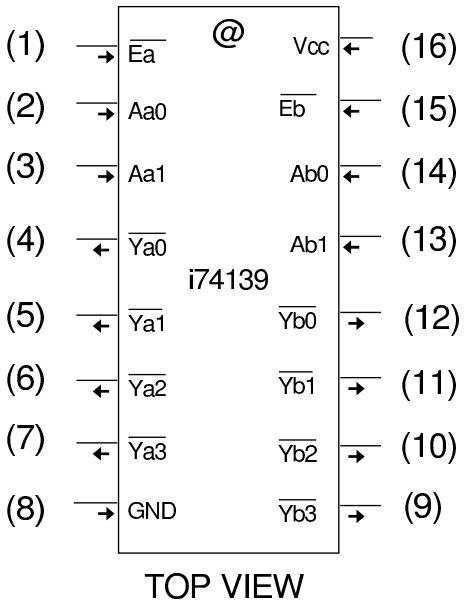
74138 Verilog code.

```
module h74138 (_Y, A, _E0, _E1, E2);
  input [2:0] A;
  input _E0, _E1, E2;
  output [7:0] _Y;
  //
  function [7:0] f74138 (input [2:0] A, input _E0,
                         input _E1, input E2);
    if (_E0 == 0 && _E1 == 0 && E2 == 1)
      begin
        case (A)
          0: f74138 = 8'b11111110;
          1: f74138 = 8'b11111101;
          2: f74138 = 8'b11111011;
          3: f74138 = 8'b11110111;
          4: f74138 = 8'b11101111;
          5: f74138 = 8'b11011111;
          6: f74138 = 8'b10111111;
```

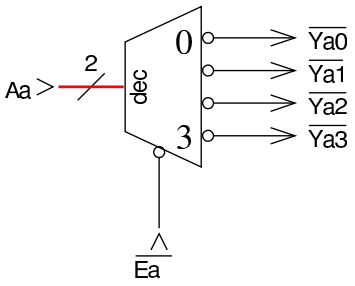
```
          7: f74138 = 8'b01111111;
        endcase
      end
    else
      begin
        f74138 = 8'b11111111;
      end
  endfunction
  //
  assign #8 _Y = f74138 (A, _E0, _E1, E2);
endmodule
```

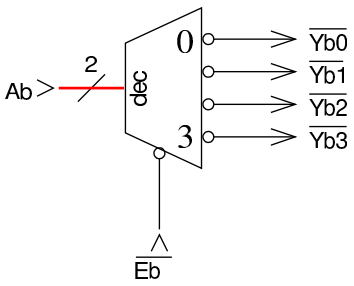
One half 74139 truth table.

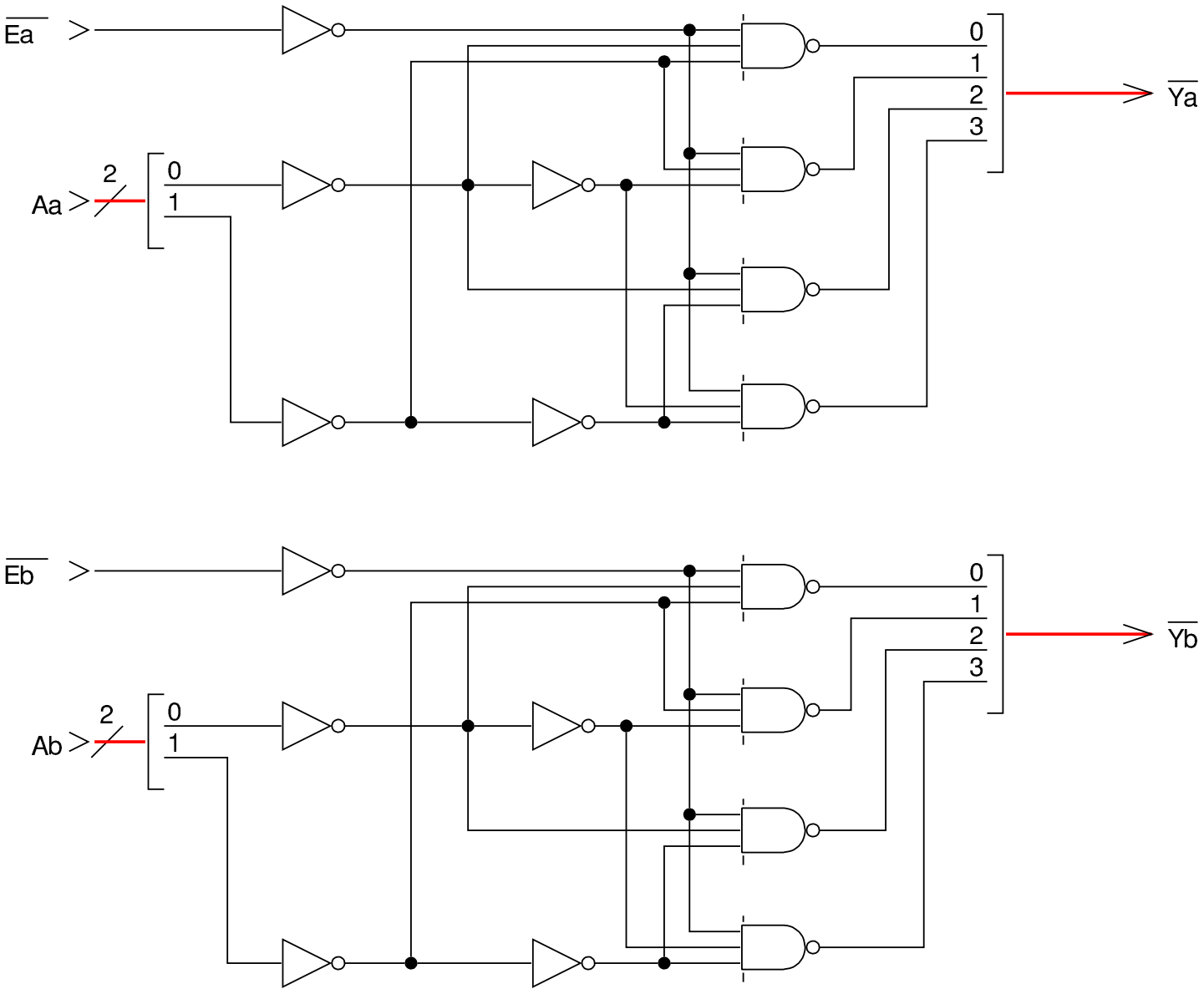
$\overline{E}$	A_1	A_0	$\overline{Y}_3$	$\overline{Y}_2$	$\overline{Y}_1$	$\overline{Y}_0$
1	X	X	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1



TOP VIEW







74139 Verilog code.

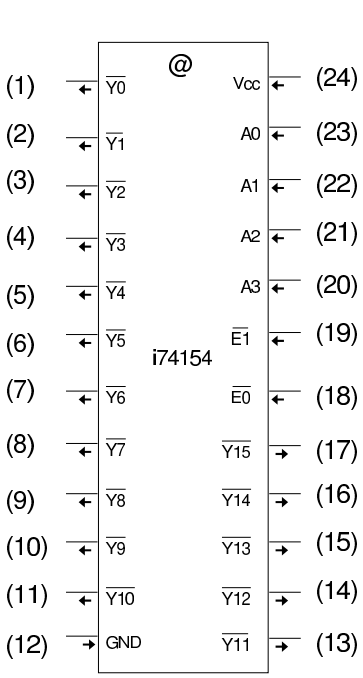
```
module h74139 (_Ya, _Yb, Aa, Ab, _Ea, _Eb);
  input [1:0] Aa, Ab;
  input _Ea, _Eb;
  output [3:0] _Ya, _Yb;
  //
  function [3:0] f74139 (input [1:0] A, input _E);
    if (_E == 0)
      begin
        case (A)
          0: f74139 = 4'b1110;
```

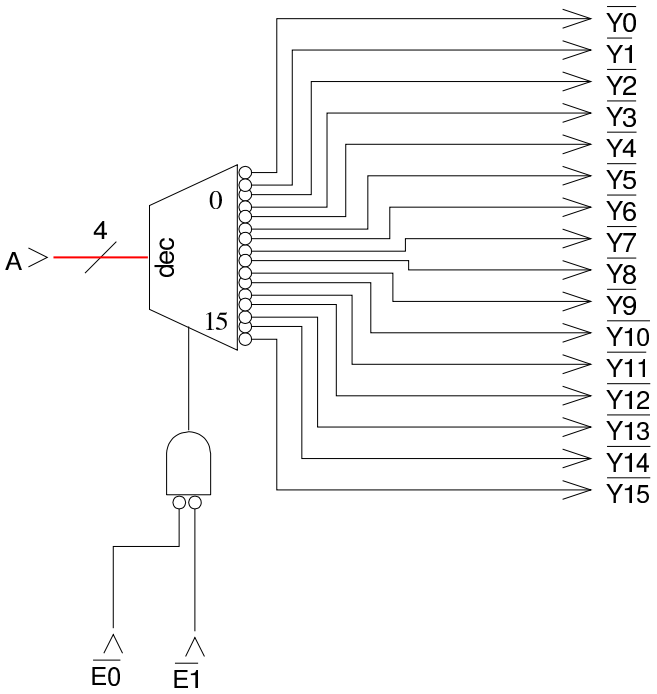
```
            1: f74139 = 4'b1101;
            2: f74139 = 4'b1011;
            3: f74139 = 4'b0111;
          endcase
        end
      else
        begin
          f74139 = 4'b1111;
        end
    endfunction
    //
    assign #8 _Ya = f74139 (Aa, _Ea);
    assign #8 _Yb = f74139 (Ab, _Eb);
endmodule
```

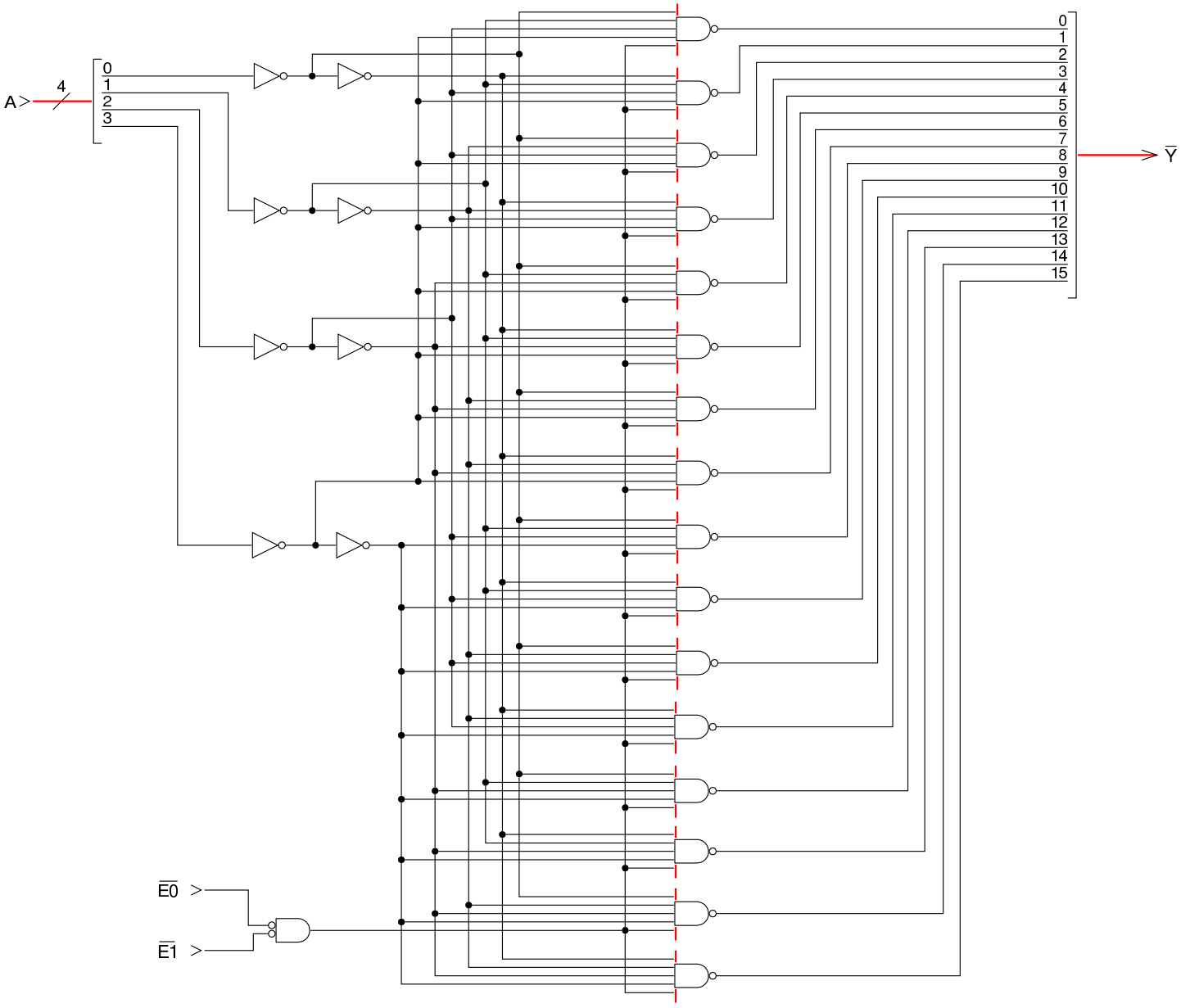
«

74154 truth table.

$\overline{E_1}$	$\overline{E_0}$	A_3	A_2	A_1	A_0	$\overline{Y_{15}}$	$\overline{Y_{14}}$	$\overline{Y_{13}}$	$\overline{Y_{12}}$	$\overline{Y_{11}}$	$\overline{Y_{10}}$	$\overline{Y_9}$	$\overline{Y_8}$	$\overline{Y_7}$	$\overline{Y_6}$	$\overline{Y_5}$	$\overline{Y_4}$	$\overline{Y_3}$	$\overline{Y_2}$	$\overline{Y_1}$	$\overline{Y_0}$
1	1	X	X	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	X	X	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	X	X	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
1	1	1	0	0	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
1	1	1	0	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1


@
(1) Y0
(2) Y1
(3) Y2
(4) Y3
(5) Y4
(6) Y5
(7) Y6
(8) Y7
(9) Y8
(10) Y9
(11) Y10
(12) GND
Vcc (24)
A0 (23)
A1 (22)
A2 (21)
A3 (20)
E1 (19)
E0 (18)
Y15 (17)
Y14 (16)
Y13 (15)
Y12 (14)
Y11 (13)
i74154


A
4
dec
0
15
E0
E1
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7
Y8
Y9
Y10
Y11
Y12
Y13
Y14
Y15



74154 Verilog code.

```
module h74154 (_Y, A, _E0, _E1);
  input [3:0] A;
  input _E0, _E1;
  output [15:0] _Y;
  //
  function [15:0] f74154 (input [3:0] A,
                          input _E0, input _E1);
    if (_E0 == 0 && _E1 == 0)
      begin
        case (A)
```

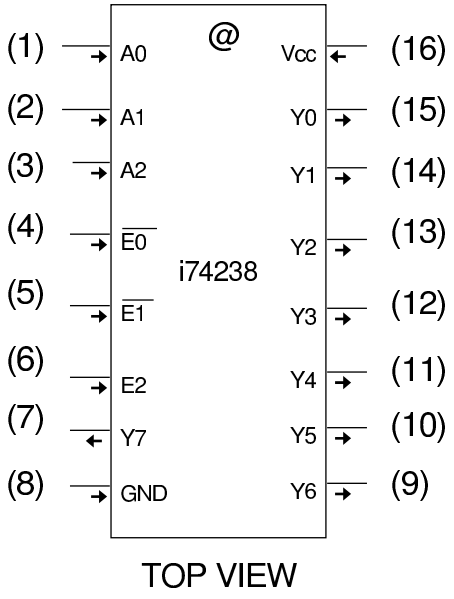
```
            0: f74154 = 16'b1111111111111110;
            1: f74154 = 16'b1111111111111101;
            2: f74154 = 16'b1111111111111011;
            3: f74154 = 16'b1111111111110111;
            4: f74154 = 16'b1111111111101111;
            5: f74154 = 16'b1111111111011111;
            6: f74154 = 16'b1111111110111111;
            7: f74154 = 16'b1111111101111111;
            8: f74154 = 16'b1111111011111111;
            9: f74154 = 16'b1111110111111111;
           10: f74154 = 16'b1111101111111111;
           11: f74154 = 16'b1111011111111111;
           12: f74154 = 16'b1110111111111111;
           13: f74154 = 16'b1101111111111111;
           14: f74154 = 16'b1011111111111111;
           15: f74154 = 16'b0111111111111111;
         endcase
       end
     else
       begin
         f74154 = 16'b1111111111111111;
       end
   endfunction
   //
   assign #8 _Y = f74154 (A, _E0, _E1);
endmodule
```

74238 3-to-8 data distributor or demultiplexer

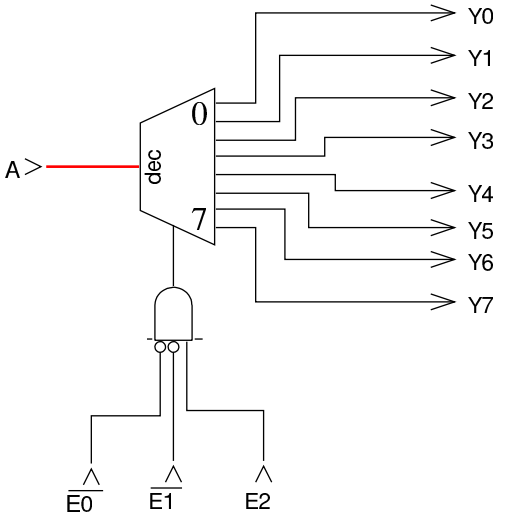
«

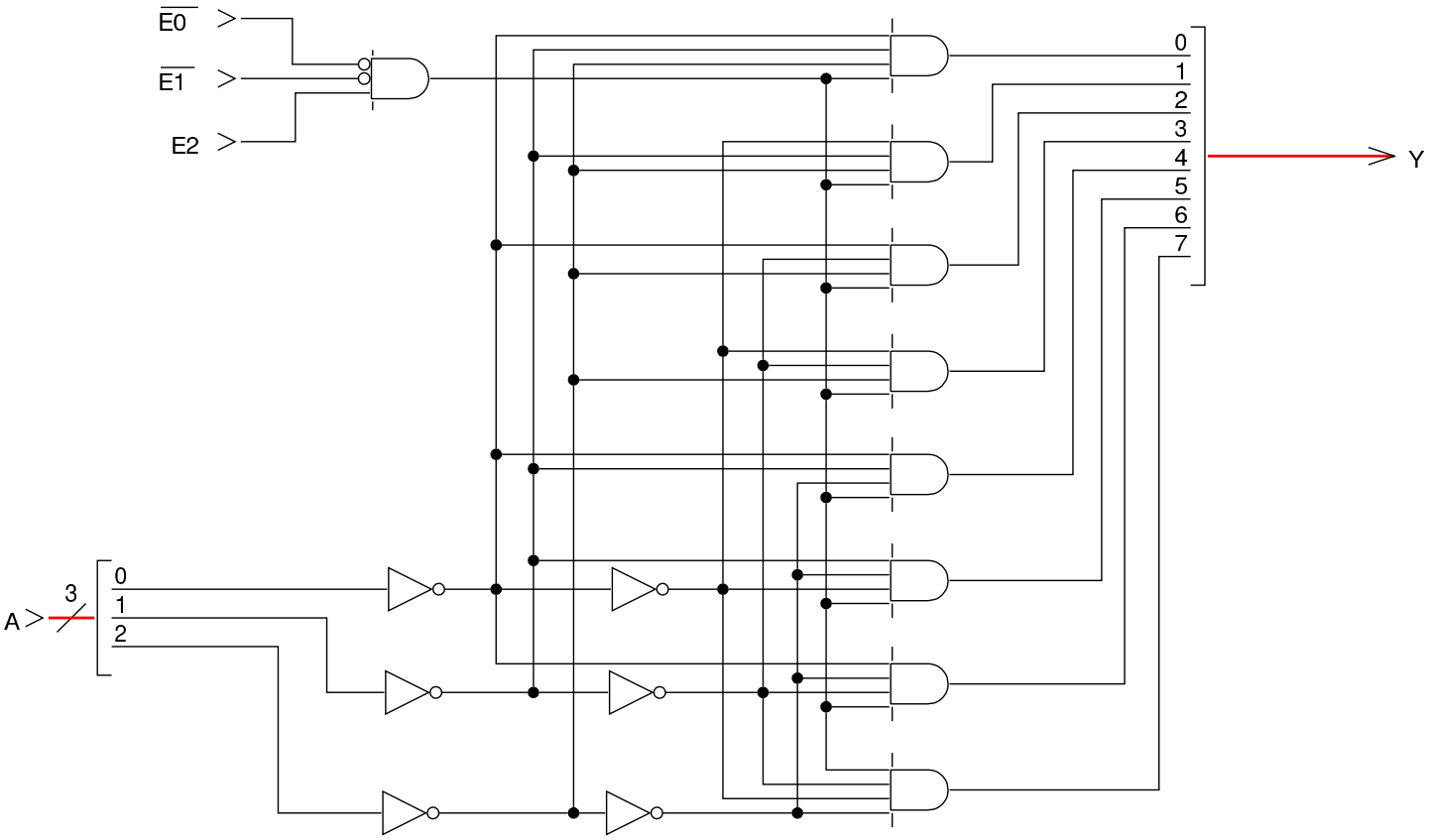
74238 truth table.

E_2	$\overline{E_1}$	$\overline{E_0}$	A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
X	X	1	X	X	X	0	0	0	0	0	0	0	0
X	1	0	X	X	X	0	0	0	0	0	0	0	0
0	X	X	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	1	0	0	0	0	0	0	1	0
1	0	0	0	1	0	0	0	0	0	0	1	0	0
1	0	0	0	1	1	0	0	0	0	1	0	0	0
1	0	0	1	0	0	0	0	0	1	0	0	0	0
1	0	0	1	0	1	0	0	1	0	0	0	0	0
1	0	0	1	1	0	0	1	0	0	0	0	0	0
1	0	0	1	1	1	1	0	0	0	0	0	0	0



TOP VIEW





74238 Verilog code.

```
module h74238 (Y, A, _E0, _E1, E2);
  input [2:0] A;
  input _E0, _E1, E2;
  output [7:0] Y;
  //
  function [7:0] f74238 (input [2:0] A,
                         input _E0, input _E1, input E2);
    if (_E0 == 0 && _E1 == 0 && E2 == 1)
      begin
        case (A)
          0: f74238 = 8'b00000001;
          1: f74238 = 8'b00000010;
          2: f74238 = 8'b00000100;
          3: f74238 = 8'b00001000;
          4: f74238 = 8'b00010000;
          5: f74238 = 8'b00100000;
          6: f74238 = 8'b01000000;
```

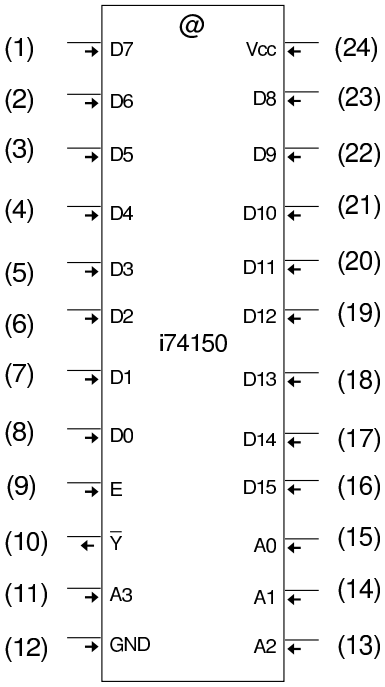
```
            7: f74238 = 8'b10000000;
          endcase
        end
      else
        begin
          f74238 = 8'b00000000;
        end
    endfunction
    //
    assign #8 Y = f74238 (A, _E0, _E1, E2);
endmodule
```

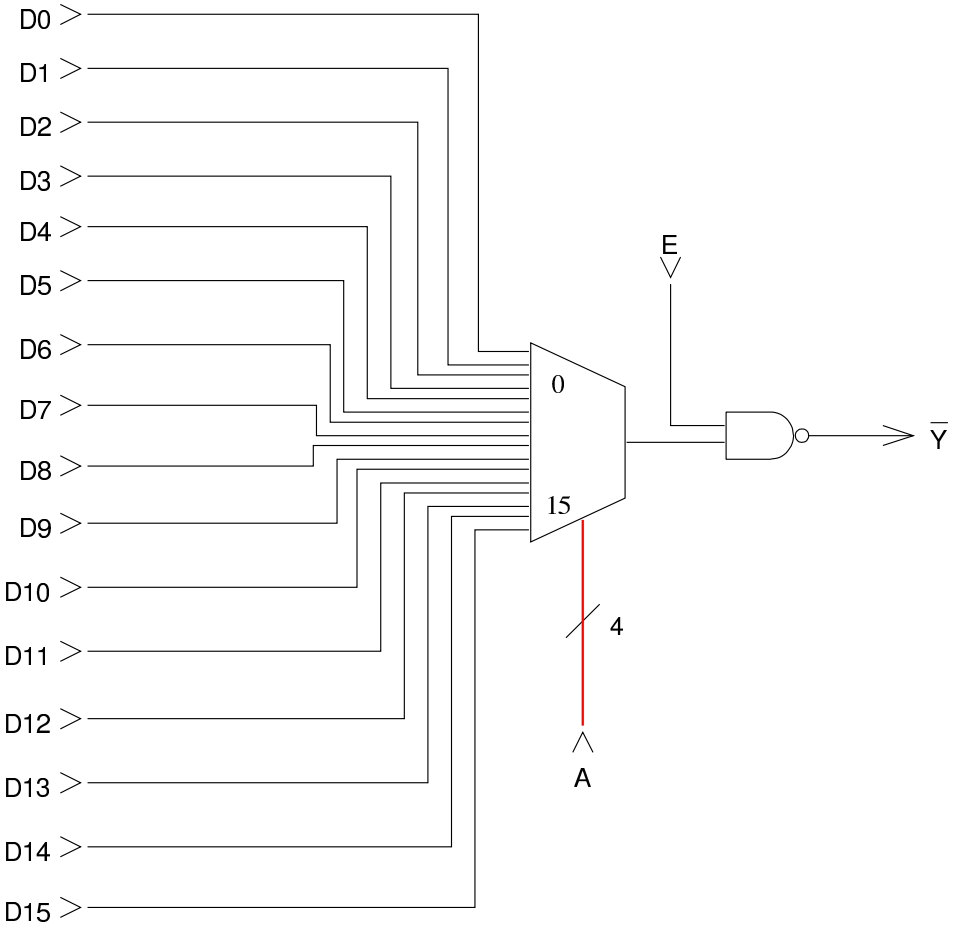
74150 1-of-16 data selector or multiplexer

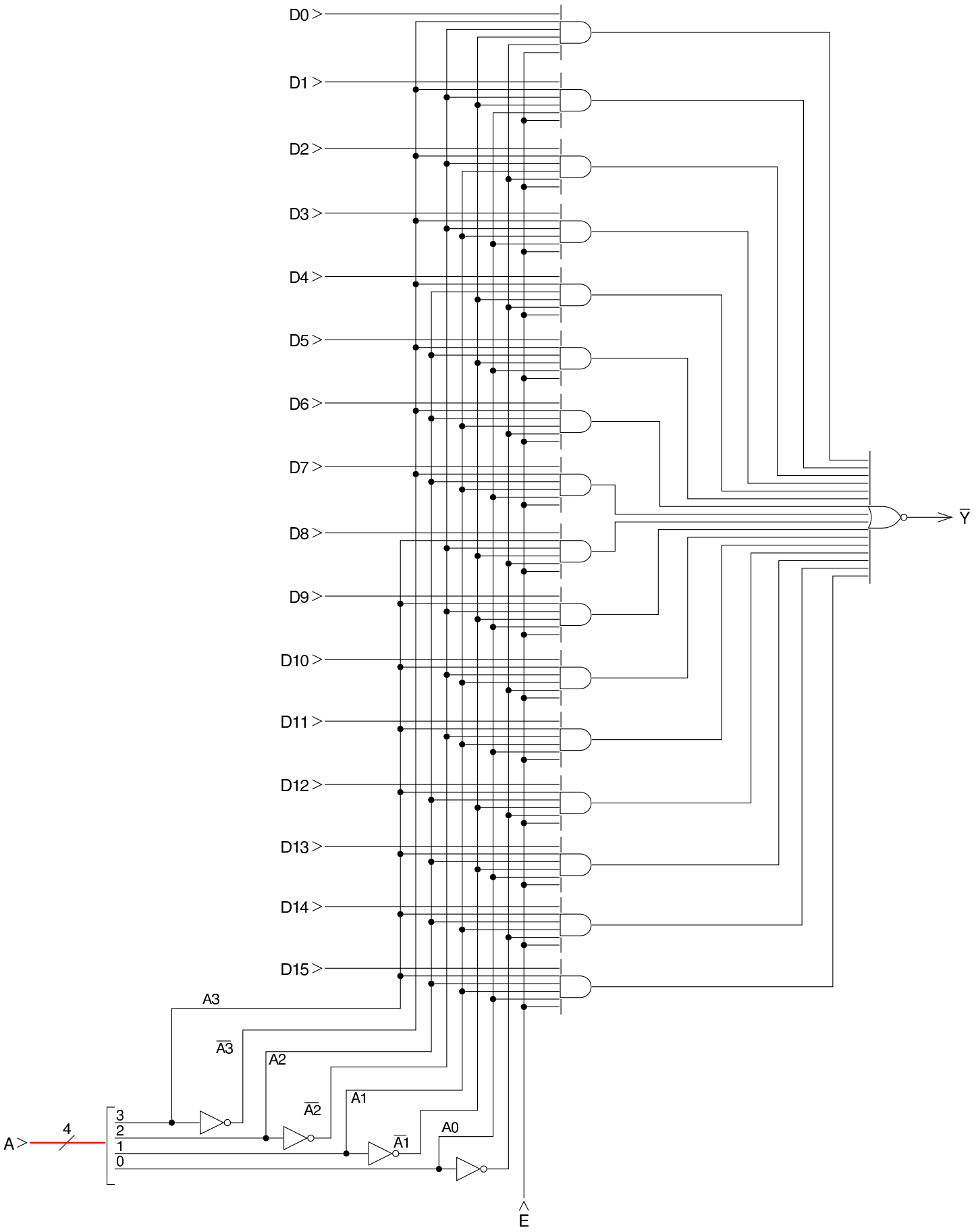
74150 truth table.

E	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_3	A_2	A_1	A_0	$\overline{Y}$
0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	0	0	0	0	1
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	0	0	0	0	0
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	X	0	0	0	1	1
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	X	0	0	0	1	0
1	X	X	X	X	X	X	X	X	X	X	X	X	X	0	X	X	0	0	1	0	1
1	X	X	X	X	X	X	X	X	X	X	X	X	X	1	X	X	0	0	1	0	0
1	X	X	X	X	X	X	X	X	X	X	X	X	0	X	X	X	0	0	1	1	1
1	X	X	X	X	X	X	X	X	X	X	X	X	1	X	X	X	0	0	1	1	0
1	X	X	X	X	X	X	X	X	X	X	X	0	X	X	X	X	0	1	0	0	1
1	X	X	X	X	X	X	X	X	X	X	X	1	X	X	X	X	0	1	0	0	0
1	X	X	X	X	X	X	X	X	X	X	0	X	X	X	X	X	0	1	0	1	1
1	X	X	X	X	X	X	X	X	X	X	1	X	X	X	X	X	0	1	0	1	0
1	X	X	X	X	X	X	X	X	X	0	X	X	X	X	X	X	0	1	1	0	1
1	X	X	X	X	X	X	X	X	X	1	X	X	X	X	X	X	0	1	1	0	0
1	X	X	X	X	X	X	X	X	0	X	X	X	X	X	X	X	0	1	1	1	1
1	X	X	X	X	X	X	X	X	1	X	X	X	X	X	X	X	0	1	1	1	0
1	X	X	X	X	X	X	X	0	X	X	X	X	X	X	X	X	1	0	0	0	1
1	X	X	X	X	X	X	X	1	X	X	X	X	X	X	X	X	1	0	0	0	0
1	X	X	X	X	X	X	0	X	X	X	X	X	X	X	X	X	1	0	0	1	1
1	X	X	X	X	X	X	1	X	X	X	X	X	X	X	X	X	1	0	0	1	0
1	X	X	X	X	X	0	X	X	X	X	X	X	X	X	X	X	1	0	1	0	1
1	X	X	X	X	X	1	X	X	X	X	X	X	X	X	X	X	1	0	1	0	0
1	X	X	X	X	0	X	X	X	X	X	X	X	X	X	X	X	1	0	1	1	1
1	X	X	X	X	1	X	X	X	X	X	X	X	X	X	X	X	1	0	1	1	0
1	X	X	X	0	X	X	X	X	X	X	X	X	X	X	X	X	1	1	0	0	1
1	X	X	X	1	X	X	X	X	X	X	X	X	X	X	X	X	1	1	0	0	0
1	X	X	0	X	X	X	X	X	X	X	X	X	X	X	X	X	1	1	0	1	1
1	X	X	1	X	X	X	X	X	X	X	X	X	X	X	X	X	1	1	0	1	0
1	X	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	1	1	0	1
1	X	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	1	1	0	0

E	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_3	A_2	A_1	A_0	$\overline{Y}$
1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	1	1	1	1
1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	1	1	1	0






D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
D14
D15
Y
A3
A3
A2
A2
A1
A1
A0
A
4
3
2
1
0
E

74150 Verilog code.

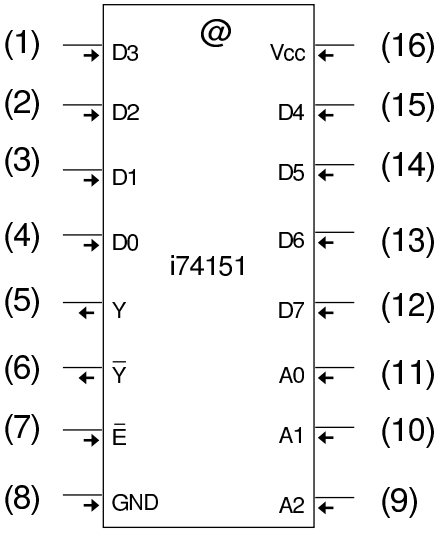
```
module h74150 (_Y, D, A, E);
  input [15:0] D;
  input [3:0] A;
  input E;
  output _Y;
  //
  function f74150 (input [15:0] D, input [3:0] A, input E);
    if (E == 1)
      begin
        case (A)
           0: f74150 = ~D[0];
           1: f74150 = ~D[1];
           2: f74150 = ~D[2];
           3: f74150 = ~D[3];
           4: f74150 = ~D[4];
           5: f74150 = ~D[5];
           6: f74150 = ~D[6];
           7: f74150 = ~D[7];
           8: f74150 = ~D[8];
           9: f74150 = ~D[9];
          10: f74150 = ~D[10];
          11: f74150 = ~D[11];
          12: f74150 = ~D[12];
          13: f74150 = ~D[13];
          14: f74150 = ~D[14];
          15: f74150 = ~D[15];
        endcase
      end
    else
      begin
        f74150 = 1;
      end
  endfunction
```

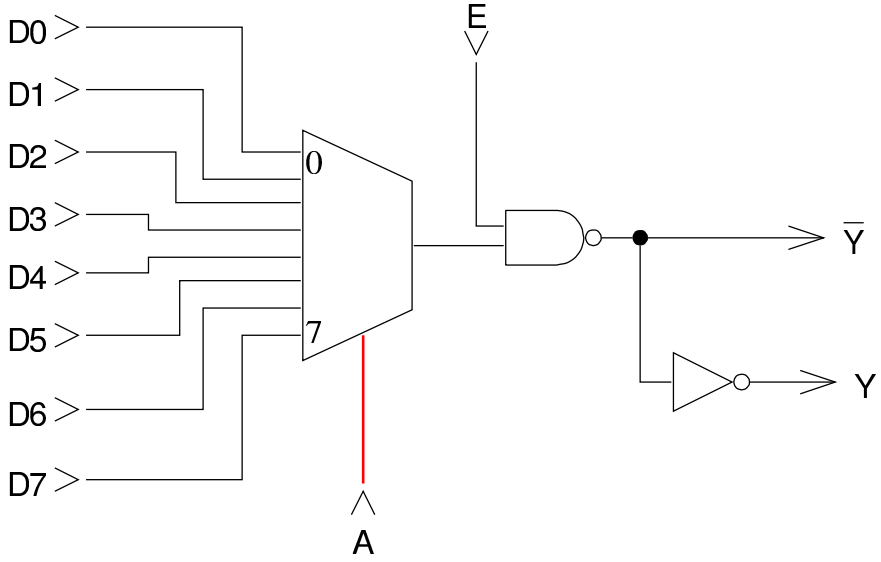
```
  //
  assign #8 _Y = f74150 (D, A, E);
endmodule
```

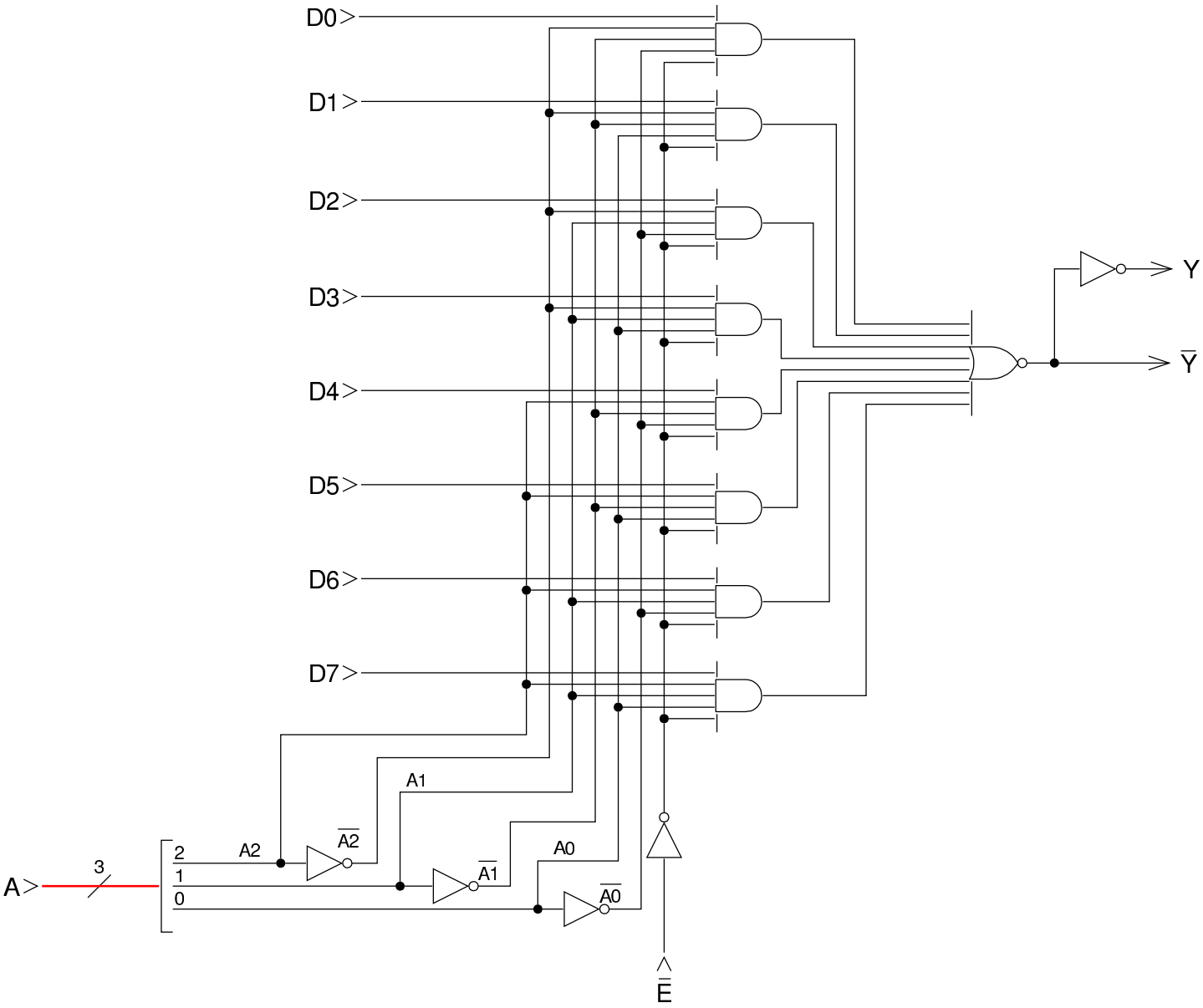
«

74151 truth table.

E	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0	$\overline{Y}$	Y
0	X	X	X	X	X	X	X	X	X	X	X	1	0
1	X	X	X	X	X	X	X	0	0	0	0	1	0
1	X	X	X	X	X	X	X	1	0	0	0	0	1
1	X	X	X	X	X	X	0	X	0	0	1	1	0
1	X	X	X	X	X	X	1	X	0	0	1	0	1
1	X	X	X	X	X	0	X	X	0	1	0	1	0
1	X	X	X	X	X	1	X	X	0	1	0	0	1
1	X	X	X	X	0	X	X	X	0	1	1	1	0
1	X	X	X	X	1	X	X	X	0	1	1	0	1
1	X	X	X	0	X	X	X	X	1	0	0	1	0
1	X	X	X	1	X	X	X	X	1	0	0	0	1
1	X	X	0	X	X	X	X	X	1	0	1	1	0
1	X	X	1	X	X	X	X	X	1	0	1	0	1
1	X	0	X	X	X	X	X	X	1	1	0	1	0
1	X	1	X	X	X	X	X	X	1	1	0	0	1
1	0	X	X	X	X	X	X	X	1	1	1	1	0
1	1	X	X	X	X	X	X	X	1	1	1	0	1







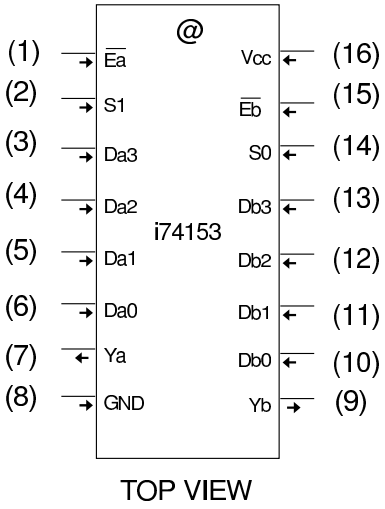
74151 Verilog code.

```
module h74151 (Y, _Y, D, A, E);
  input [7:0] D;
  input [2:0] A;
  input E;
  output Y, _Y;
  //
  function f74151 (input [7:0] D, input [2:0] A, input E);
    if (E == 1)
      begin
        case (A)
```

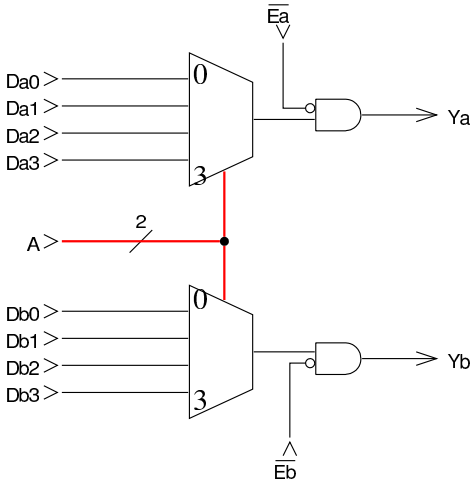
```
            0: f74151 = D[0];
            1: f74151 = D[1];
            2: f74151 = D[2];
            3: f74151 = D[3];
            4: f74151 = D[4];
            5: f74151 = D[5];
            6: f74151 = D[6];
            7: f74151 = D[7];
          endcase
        end
      else
        begin
          f74151 = 0;
        end
    endfunction
    //
    assign #8 Y  = f74151 (D, A, E);
    assign #8 _Y = ~Y;
endmodule
```

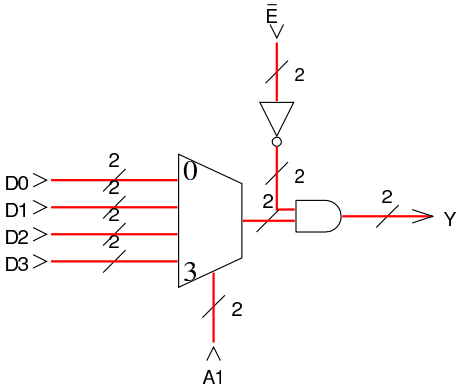
half 74153 truth table.

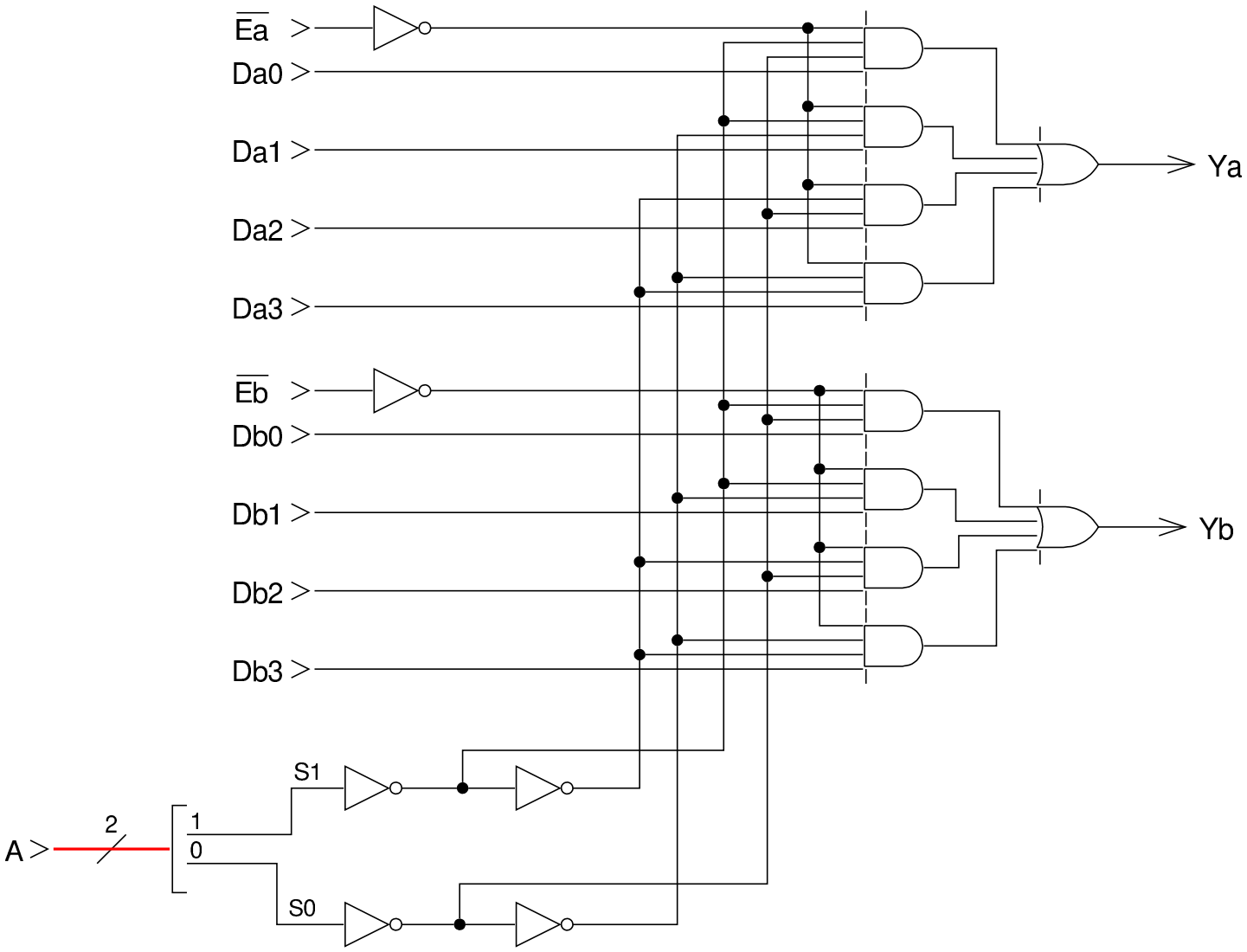
$\overline{E}$	D_3	D_2	D_1	D_0	A_1	A_0	Y
1	X	X	X	X	X	X	0
0	X	X	X	0	0	0	0
0	X	X	X	1	0	0	1
0	X	X	0	X	0	1	0
0	X	X	1	X	0	1	1
0	X	0	X	X	1	0	0
0	X	1	X	X	1	0	1
0	0	X	X	X	1	1	0
0	1	X	X	X	1	1	1



TOP VIEW







74153 Verilog code.

```
module h74153 (Ya, Yb, Da, Db, A, _Ea, _Eb);
  input [3:0] Da, Db;
  input [1:0] A;
  input _Ea, _Eb;
  output Ya, Yb;
  //
  function f74153 (input [3:0] D, input [1:0] A, input _E);
    if (_E == 0)
      begin
        case (A)
          0: f74153 = D[0];
          1: f74153 = D[1];
```

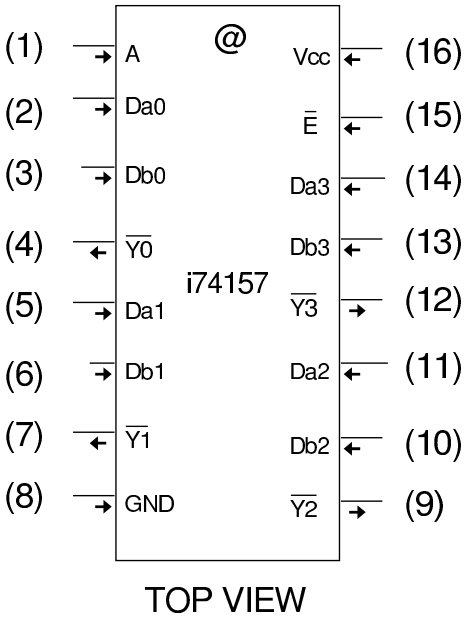
```
            2: f74153 = D[2];
            3: f74153 = D[3];
          endcase
        end
      else
        begin
          f74153 = 0;
        end
    endfunction
    //
    assign #8 Ya = f74153 (Da, A, _Ea);
    assign #8 Yb = f74153 (Db, A, _Eb);
endmodule
```

74157 quad 1-of-2 data selector or multiplexer

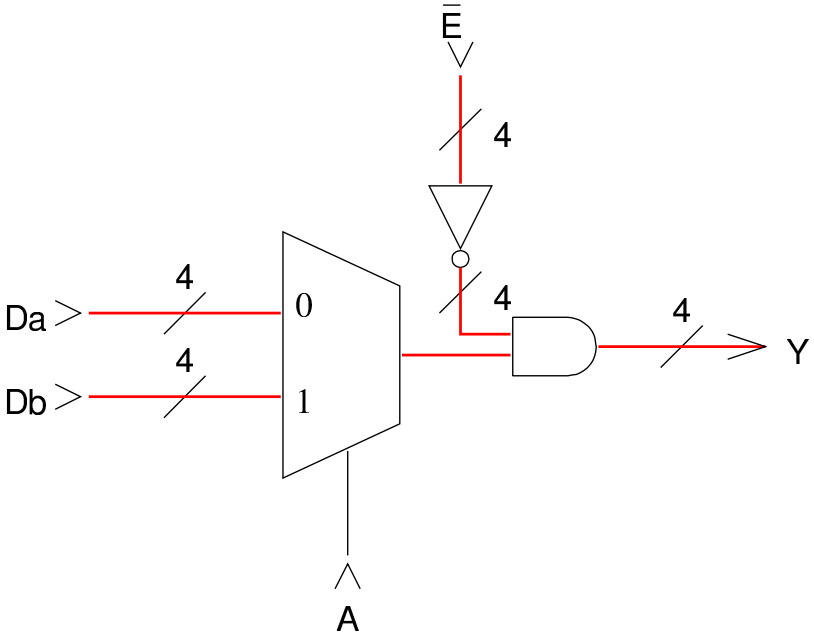
«

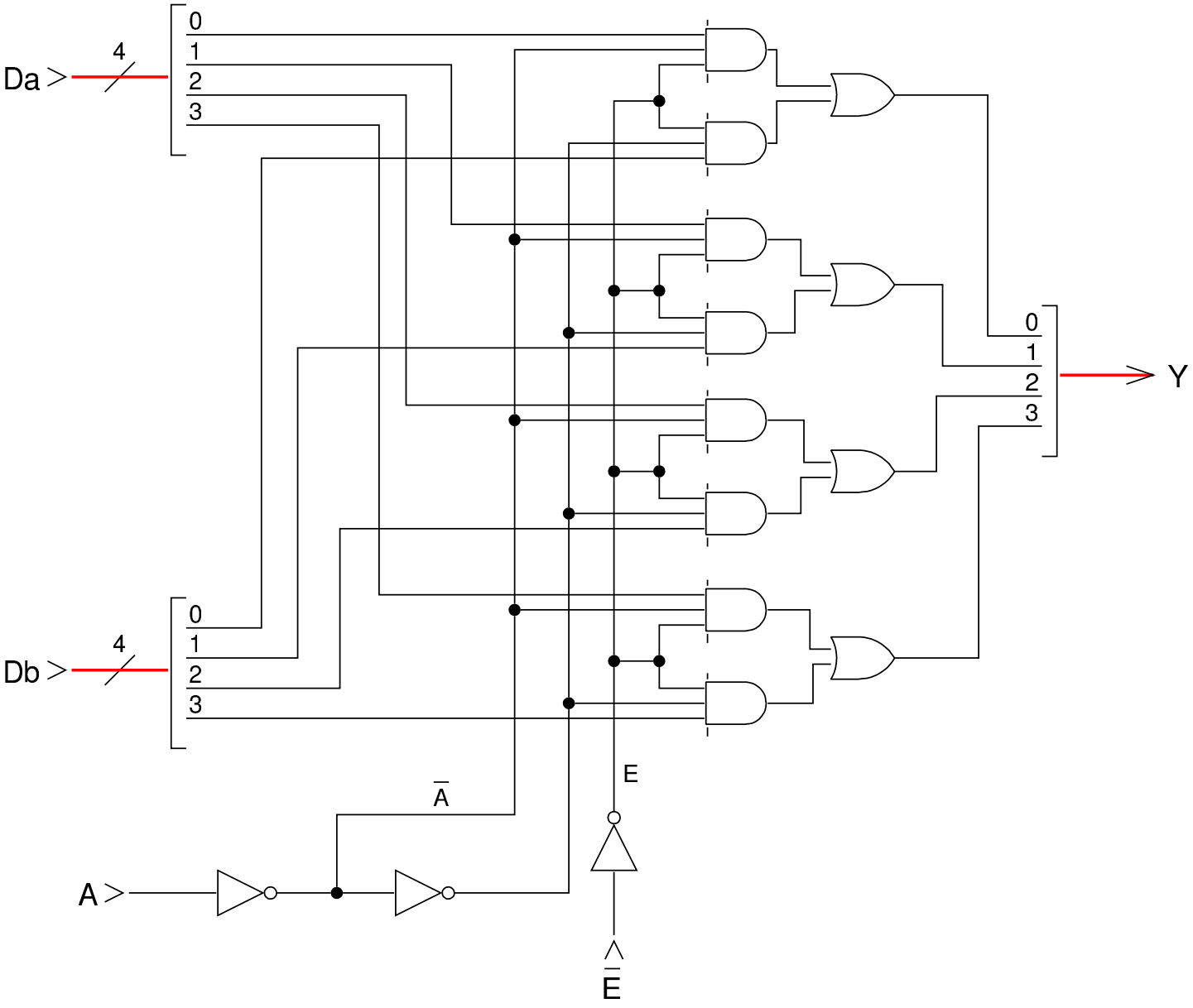
74157 truth table.

$\overline{E}$	Da_n	Db_n	A	Y_n
1	X	X	X	0
0	X	0	0	0
0	X	1	0	1
0	0	X	1	0
0	1	X	1	1



TOP VIEW





74157 Verilog code.

```
module h74157 (Y, Da, Db, A, _E);
  input [3:0] Da, Db;
  input A;
  input _E;
  output [3:0] Y;
  //
  function [3:0] f74157 (input [3:0] Da, input [3:0] Db,
                         input A, input _E);
    if (_E == 0)
      begin
```

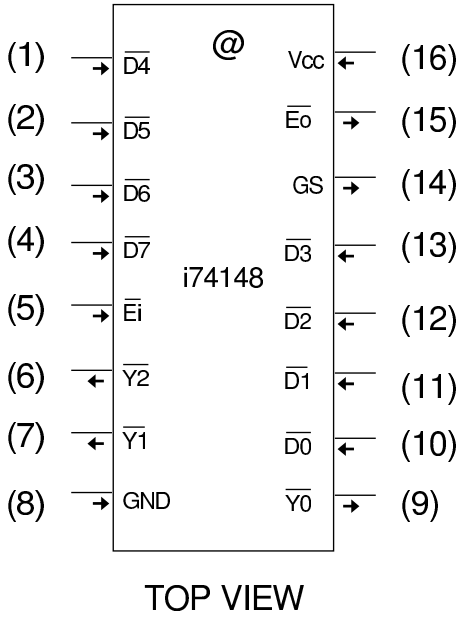
```
        case (A)
          0: f74157 = Da;
          1: f74157 = Db;
        endcase
      end
    else
      begin
        f74157 = 0;
      end
  endfunction
  //
  assign #8 Y = f74157 (Da, Db, A, _E);
endmodule
```

74148 8-to-3 priority encoder

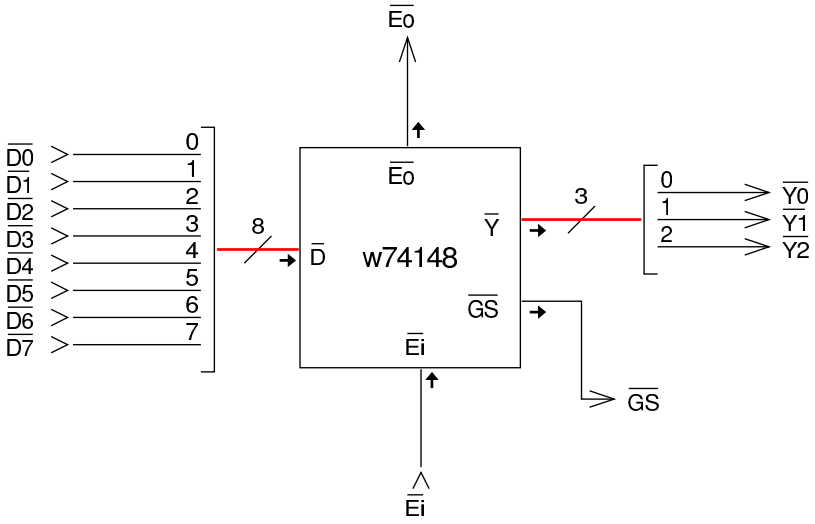
The 74148 is a priority encoder with negated inputs and outputs. It has an enable input and an enable output, used to activate the encoder and to allow concatenation with other 74148 modules. The output selects the higher input index and the module concatenation starts from the higher input module, as the last picture shows. The output $\overline{GS}$ (*group select*) is asserted if at least an input $\overline{D_n}$ is asserted.

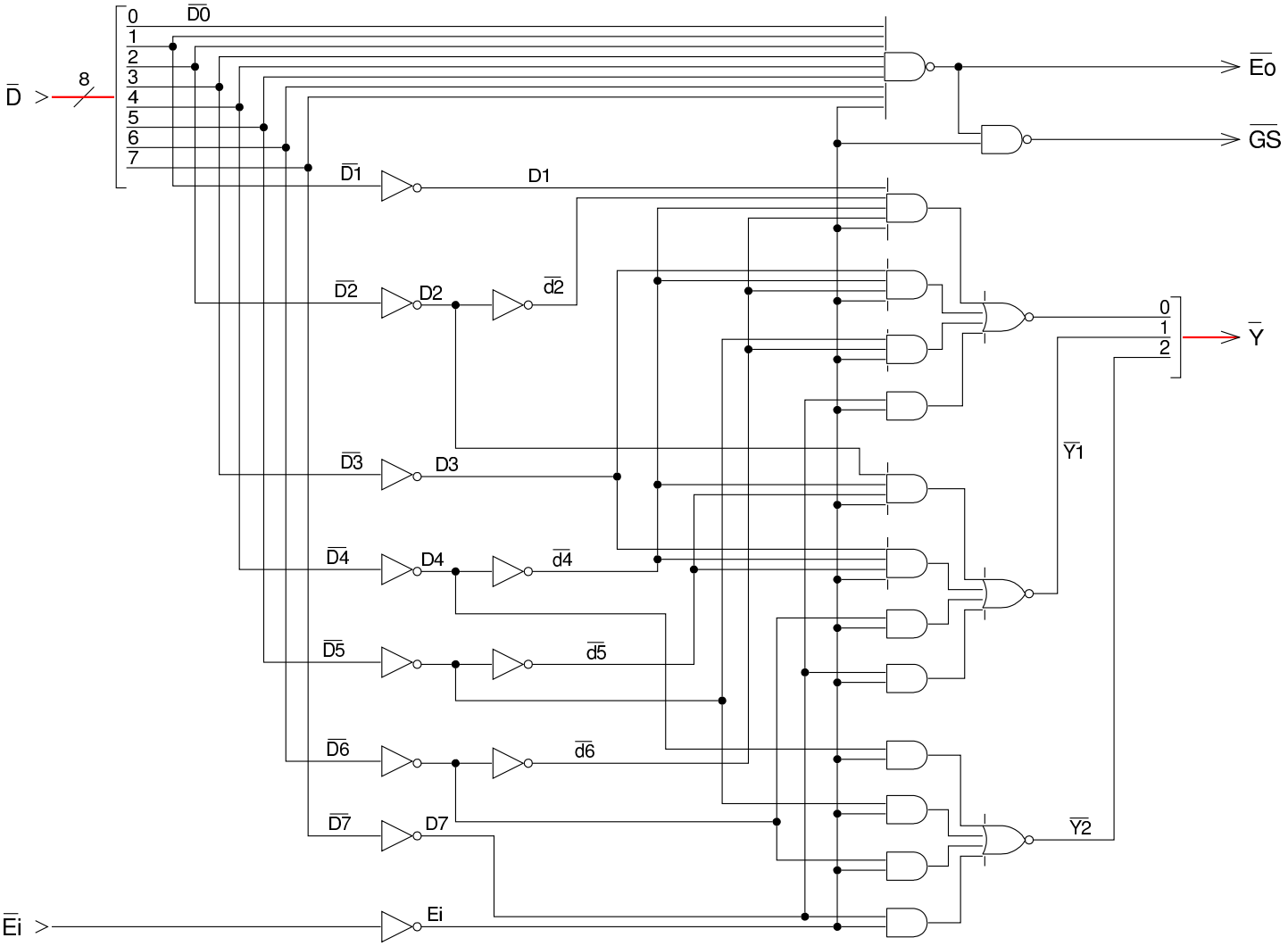
74148 truth table.

$\overline{Ei}$	$\overline{D_7}$	$\overline{D_6}$	$\overline{D_5}$	$\overline{D_4}$	$\overline{D_3}$	$\overline{D_2}$	$\overline{D_1}$	$\overline{D_0}$	$\overline{GS}$	$\overline{Eo}$	$\overline{Y_2}$	$\overline{Y_1}$	$\overline{Y_0}$
1	X	X	X	X	X	X	X	X	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	0	1	1	1
0	0	X	X	X	X	X	X	X	0	1	0	0	0
0	1	0	X	X	X	X	X	X	0	1	0	0	1
0	1	1	0	X	X	X	X	X	0	1	0	1	0
0	1	1	1	0	X	X	X	X	0	1	0	1	1
0	1	1	1	1	0	X	X	X	0	1	1	0	0
0	1	1	1	1	1	0	X	X	0	1	1	0	1
0	1	1	1	1	1	1	0	X	0	1	1	1	0
0	1	1	1	1	1	1	1	0	0	1	1	1	1



TOP VIEW





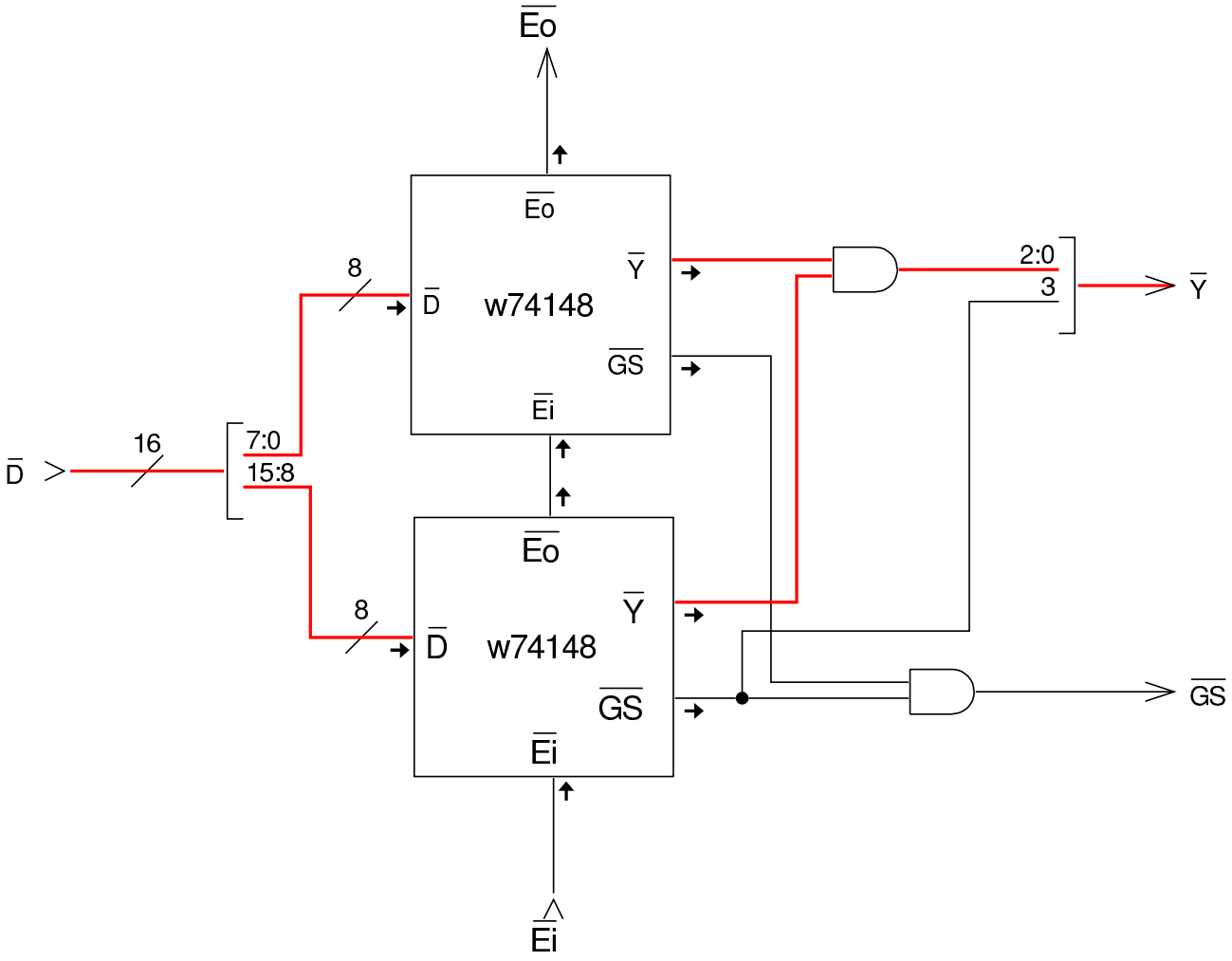
74148 Verilog code.

```
module h74148 (_GS, _Eo, _Y, _Ei, _D);
  input _Ei;
  input [7:0] _D;
  output _GS, _Eo;
  output [2:0] _Y;
  //
  function [2:0] f74148y (input _Ei, input [7:0] _D);
    if (_Ei == 0)
      begin
        if (~_D & 8'b10000000)
          f74148y = 3'b000;
        else if (~_D & 8'b01000000)
          f74148y = 3'b001;
```

```
        else if (~_D & 8'b00100000)
          f74148y = 3'b010;
        else if (~_D & 8'b00010000)
          f74148y = 3'b011;
        else if (~_D & 8'b00001000)
          f74148y = 3'b100;
        else if (~_D & 8'b00000100)
          f74148y = 3'b101;
        else if (~_D & 8'b00000010)
          f74148y = 3'b110;
        else if (~_D & 8'b00000001)
          f74148y = 3'b111;
        else
          f74148y = 3'b111;
      end
    else
      f74148y = 3'b111;
  endfunction
  function f74148gs (input _Ei, input [7:0] _D);
    if (_Ei == 1)
      f74148gs = 1;
    else if (_D == 8'b11111111)
      f74148gs = 1;
    else
      f74148gs = 0;
  endfunction
  function f74148eo (input _Ei, input [7:0] _D);
    if (_Ei == 1)
      f74148eo = 1;
    else if (_Ei == 0 && _D == 8'b11111111)
      f74148eo = 0;
    else
      f74148eo = 1;
  endfunction
```

```
  //
  assign #8 _Y  = f74148y (_Ei, _D);
  assign #8 _GS = f74148gs (_Ei, _D);
  assign #8 _Eo = f74148eo (_Ei, _D);
endmodule
```

A chain of two 74148.

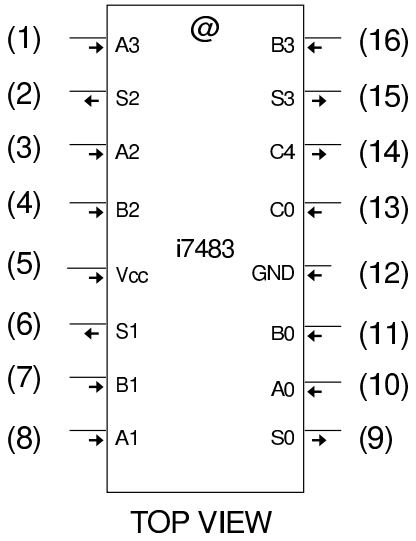


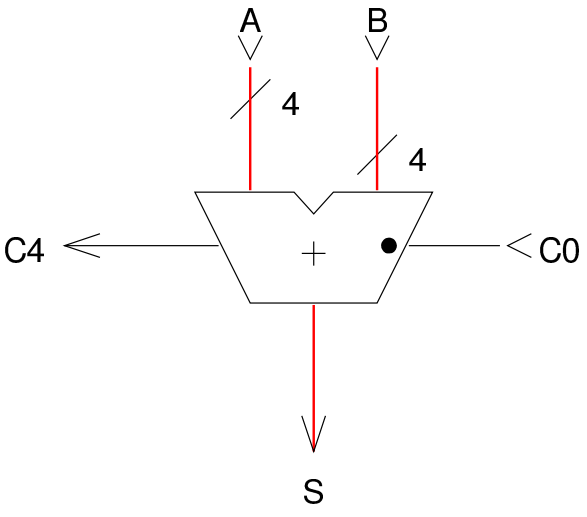
7483 4-bit binary full-adder

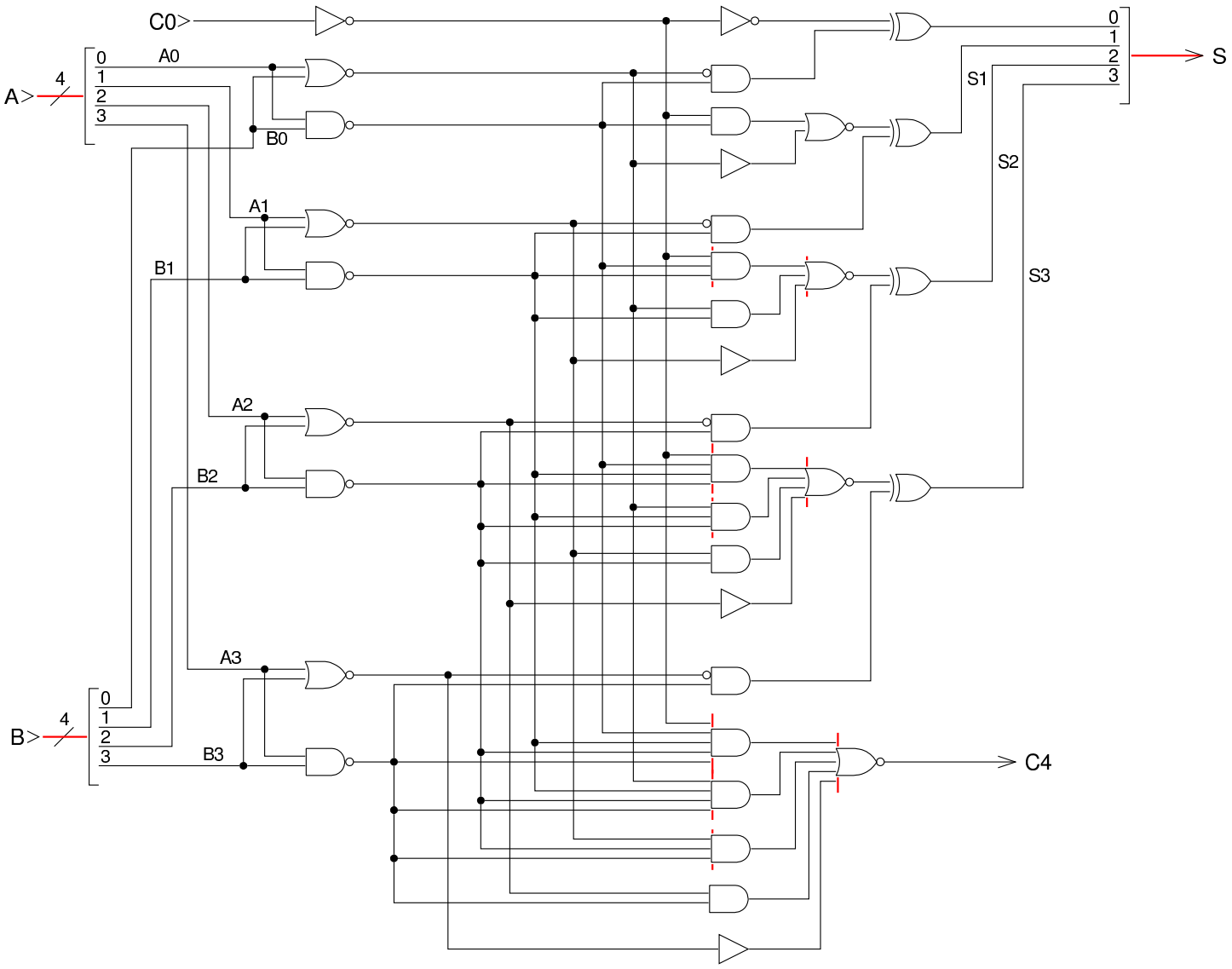
The 7483 is a 4-bit binary full-adder with carry-lookahead. The input carry is labeled C_0, whereas the output carry is C_4.

7483 truth table.

A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	C_0	C_4	S_3	S_2	S_1	S_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	1	1	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	0	0	1	0	1	0	0	0	1	1
0	0	0	0	0	0	1	1	0	0	0	0	1	1
...				...					...				
0	0	0	0	1	1	1	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	1	0	0	0	1
0	0	1	0	1	1	1	1	1	1	0	0	1	0
0	0	1	1	1	1	1	1	1	1	0	0	1	1
...				...					...				
1	1	0	0	1	1	1	1	1	1	1	1	0	0
1	1	0	1	1	1	1	1	1	1	1	1	0	1
1	1	1	0	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1



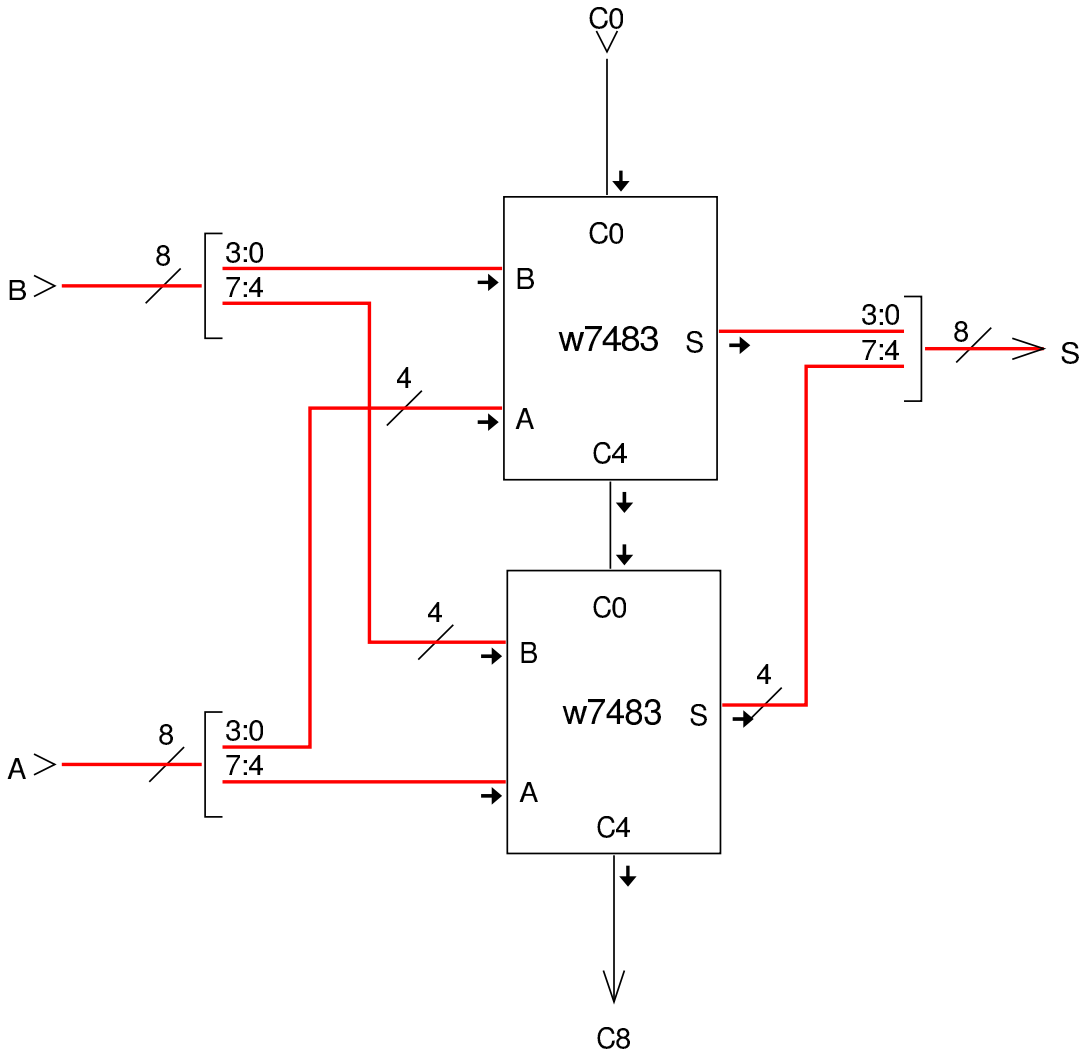




7483 Verilog code.

```
module h7483 (C4, S, A, B, C0);
  input [3:0] A, B;
  input C0;
  output C4;
  output [3:0] S;
  //
  assign #12 {C4, S} = A + B + C0;
endmodule
```

A chain of two 7483.



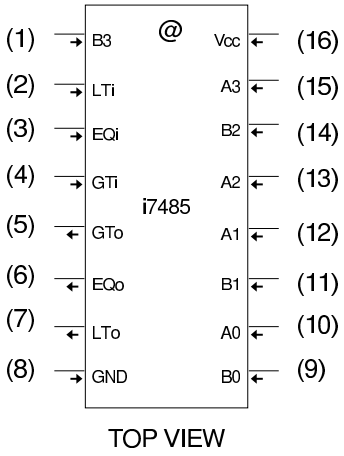
7485 4-bit magnitude comparator

«

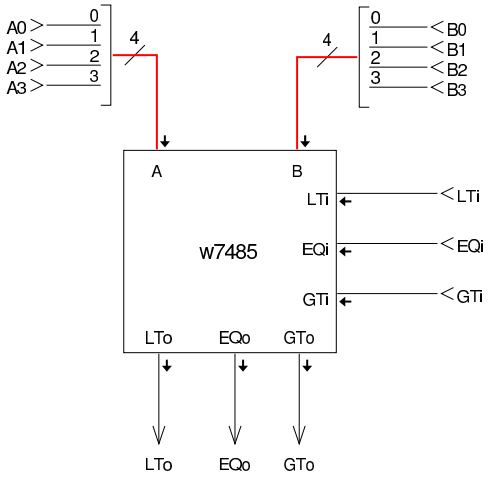
The 7485 is a 4-bit magnitude comparator that can be extended with other 7485 to increase the bit-size comparison. First are compared the inputs ***A*** and ***B***, then, if they are equal, the values form ***AB0***, ***BA0*** and ***EQ0*** are used. The inputs ***AB0***, ***BA0*** and ***EQ0*** can be used to connect another 7485 which compares less significant digits.

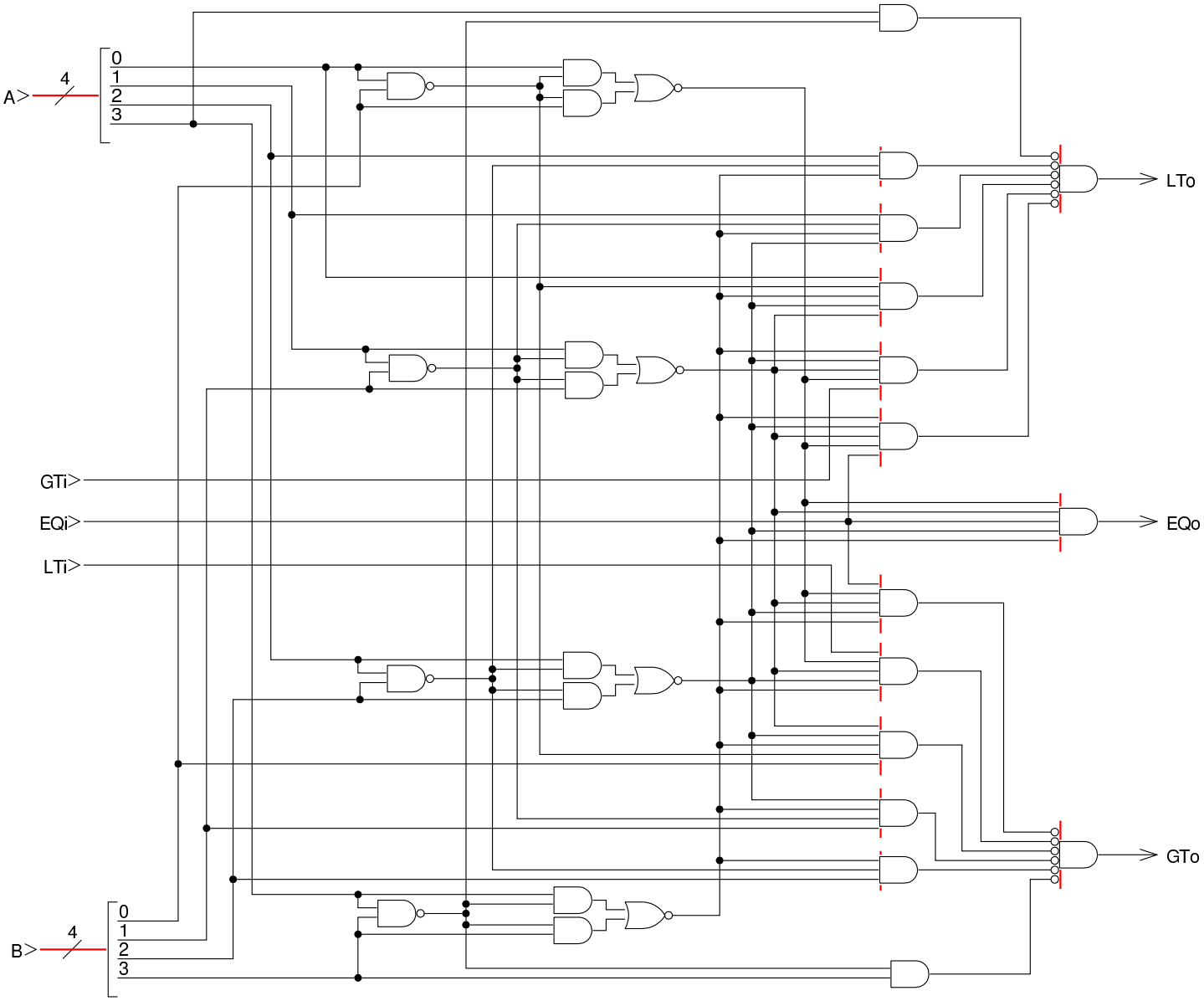
7485 truth table.

A and ***B***	***GTi***	***EQi***	***LTi***	***GTo***	***EQo***	***LTo***
A=B	0	0	0	1	0	1
A=B	0	0	1	0	0	1
A=B	0	1	0	0	1	0
A=B	0	1	1	0	1	0
A=B	1	0	0	1	0	0
A=B	1	0	1	0	0	0
A=B	1	1	0	0	1	0
A=B	1	1	1	0	1	0
A>B	X	X	X	1	0	0
A<B	X	X	X	0	0	1



TOP VIEW



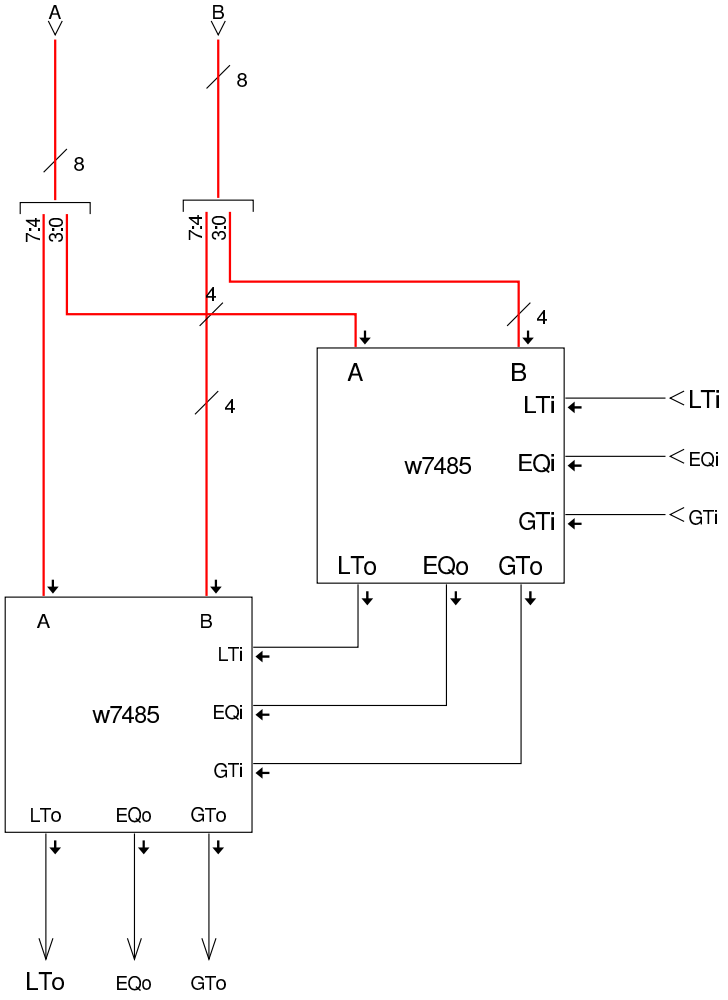


7485 Verilog code: the output ports ***GTo***, ***EQo*** and ***LTo*** do not react exactly in the same way when ***A*** and ***B*** are equal and the input ports ***GTi***, ***EQi*** and ***LTi*** have incoherent values.

```
module h7485 (GTo, EQo, LTo, A, B, GTi, EQi, LTi);
  input [3:0] A, B;
  input GTi, EQi, LTi;
  output GTo, EQo, LTo;
  wire GT, EQ, LT;
  //
  assign #6 GT = A > B;
  assign #6 EQ = A == B;
```

```
  assign #6 LT = A < B;
  //
  assign #6 GTo = EQ?GTi:GT;
  assign #6 EQo = EQ?EQi:EQ;
  assign #6 LTo = EQ?LTi:LT;
endmodule
```

A chain of two 7485.



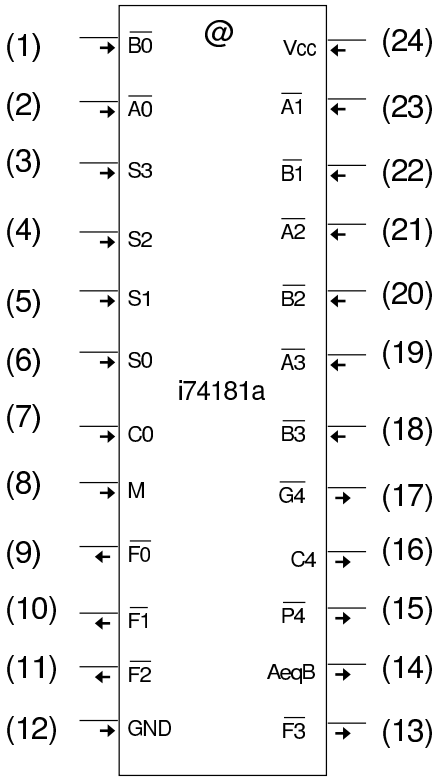
The 74181 can work in four different modes, but not all of them have properly calculated values for the ***G*** and ***P*** ports, which should be used by a carry lookahead generator. Only two of the four working modes are taken into consideration in this chapter and for this one the ports ***A***, ***B***, ***F***, ***G*** and ***P*** are "active low": in this mode the output ports required by a carry lookahead generator are correctly working as expected.[1]

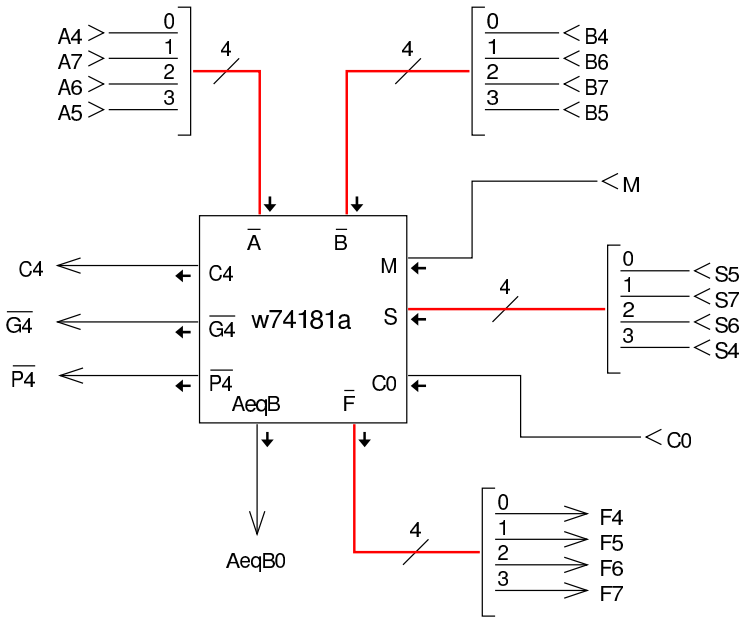
The 74181 has the ***AeqB*** output port that is almost useless: it becomes active only when the ***F*** (or $\overline{F}$) output is equal to 1111_2; on the other hand, this ALU does not have an output for the overflow condition.

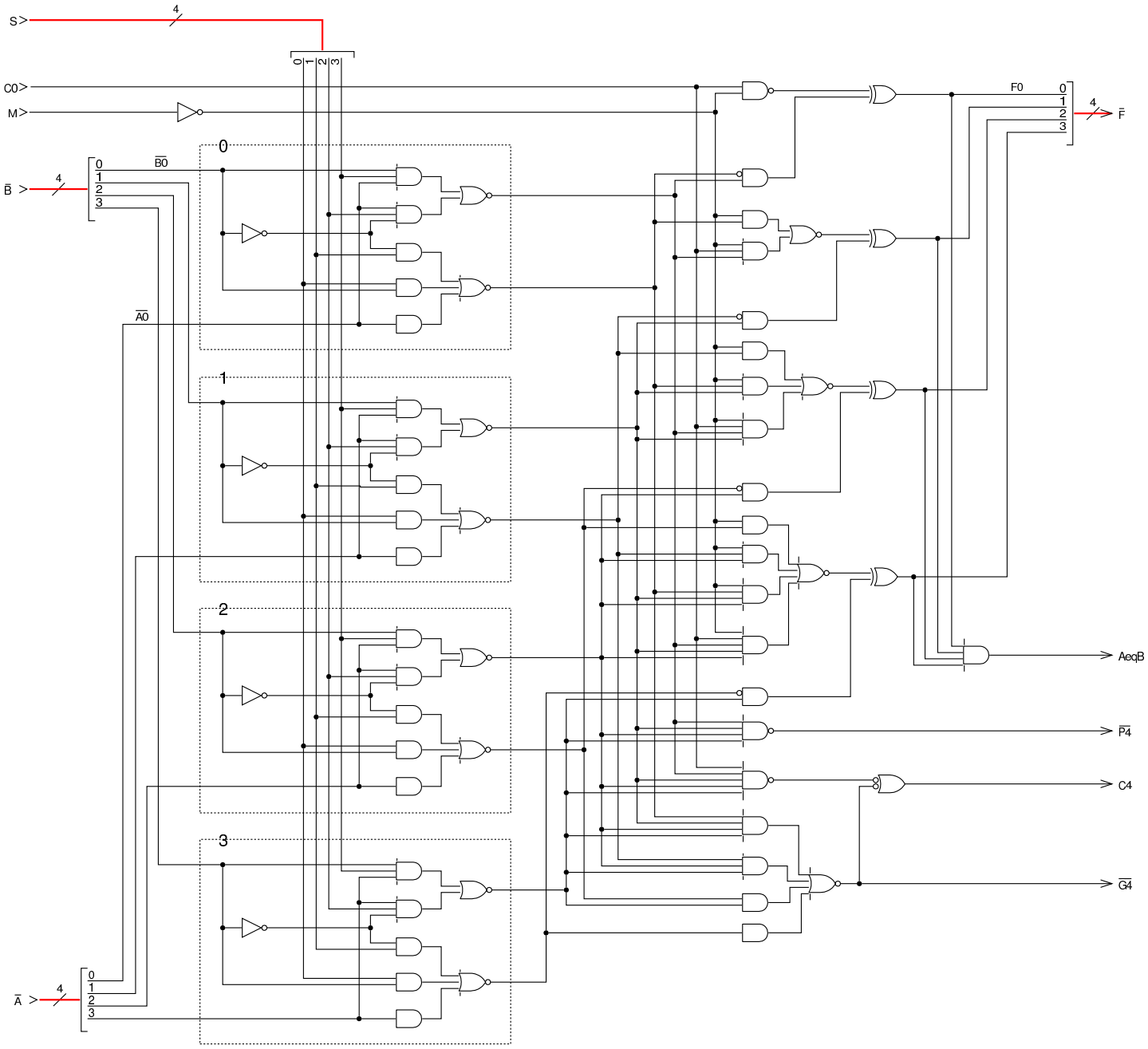
74181 operation table.

$S_{3..0}$	M	C_0	C_4	$\overline{F}_{3..0}$	$\overline{P}$	$\overline{G}$
0000	0	0		A		
0000	0	1		A minus 1		
0000	1	X		$\overline{A}$		
0001	0	0		$A \cdot B$		
0001	0	1		$(A \cdot B)$ minus 1		
0001	1	X		$\overline{A} \cdot \overline{B}$		
0010	0	0		$A \cdot \overline{B}$		
0010	0	1		$(A \cdot \overline{B}$ minus 1		
0010	1	X		$\overline{A} + B$		
0011	0	0		0000	0	1
0011	0	1		1111	1	1
0011	1	0		0000	0	1
0011	1	1		0000	1	1
0100	0	0		A plus $(A + \overline{B})$ plus 1		
0100	0	1		A plus $(A \cdot \overline{B})$		
0100	1	X		$\overline{(A + B)}$		
0101	0	0		$A + \overline{B}$ plus $(A \cdot B)$ plus 1		
0101	0	1		$A + \overline{B}$ plus $(A \cdot B)$		
0101	1	X		$\overline{B}$		
0110	0	0		A minus B		
0110	0	1		A minus B minus 1		
0110	1	X		$\overline{(A \oplus B)}$		
0111	0	0		$A + \overline{B}$ plus 1		
0111	0	1		$A + \overline{B}$		

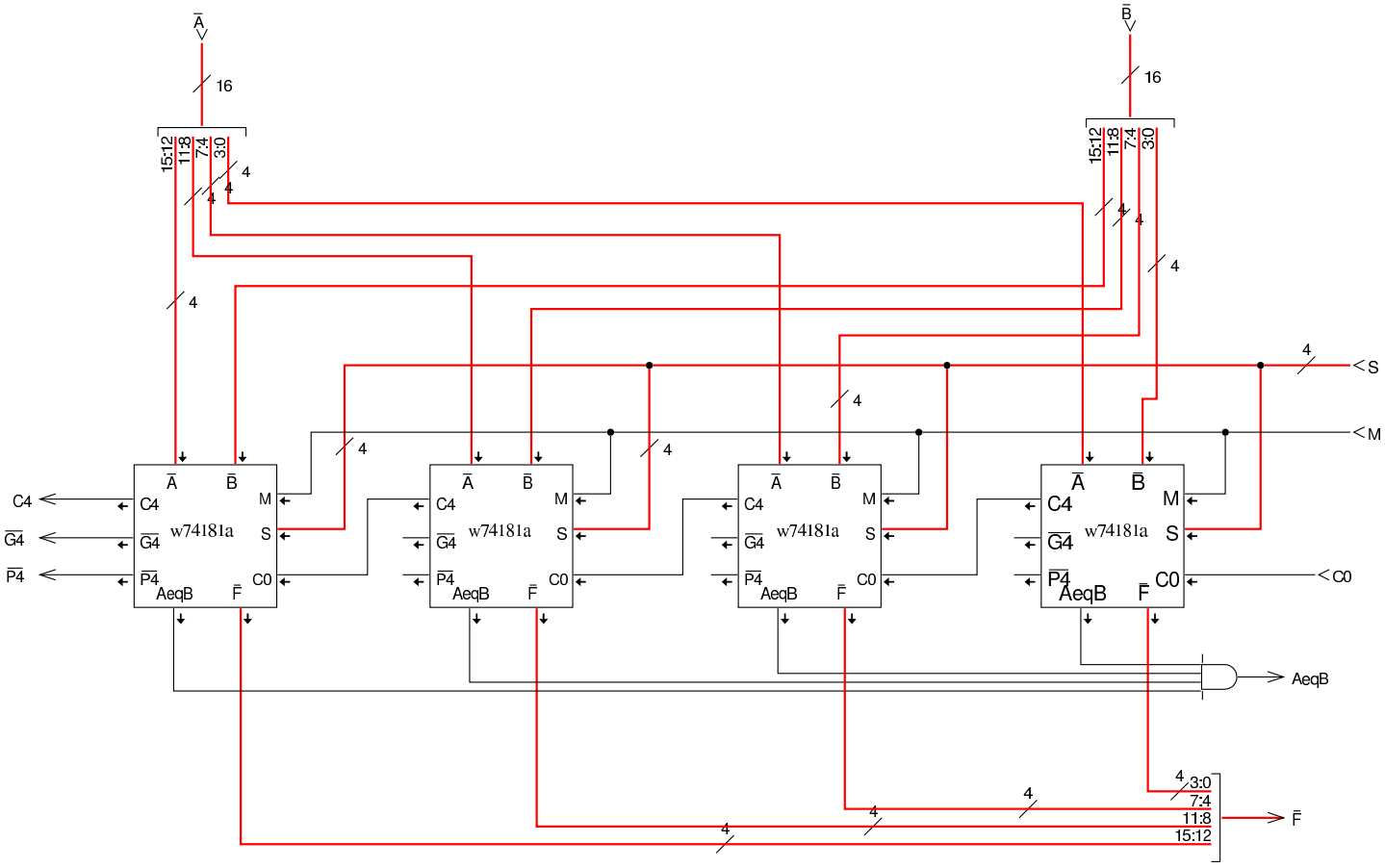
$S_{3..0}$	M	C_0	C_4	$\overline{F}_{3..0}$	$\overline{P}$	$\overline{G}$
0111	1	X		$A+\overline{B}$		
1000	0	0		A plus $(A+B)$ plus 1		
1000	0	1		A plus $(A+B)$		
1000	0	X		$\overline{A}\cdot B$		
1001	0	0		A plus B plus 1		
1001	0	1		A plus B		
1001	1	X		$A \oplus B$		
1010	0	0		$(A+B)$ plus $(A\cdot\overline{B})$ plus 1		
1010	0	1		$(A+B)$ plus $(A.\overline{B})$		
1010	1	X		B		
1011	0	0		$A+B$ plus 1		
1011	0	1		$A+B$		
1011	1	X		$A+B$		
1100	0	0		A plus A plus 1		
1100	0	1		A plus A		
1100	1	X		1111		
1101	0	0		A plus $(A\cdot B)$ plus 1		
1101	0	1		A plus $(A\cdot B)$		
1101	1	X		$A\cdot\overline{B}$		
1110	0	0		A plus $(A\cdot\overline{B})$ plus 1		
1110	0	1		A plus $(A\cdot\overline{B})$		
1110	1	X		$A\cdot B$		
1111	0	0		A plus 1		
1111	0	0		A		
1111	1	X		A		


@
i74181a
(1) B0
(2) A0
(3) S3
(4) S2
(5) S1
(6) S0
(7) C0
(8) M
(9) F0
(10) F1
(11) F2
(12) GND
Vcc (24)
A1 (23)
B1 (22)
A2 (21)
B2 (20)
A3 (19)
B3 (18)
G4 (17)
C4 (16)
P4 (15)
AeqB (14)
F3 (13)

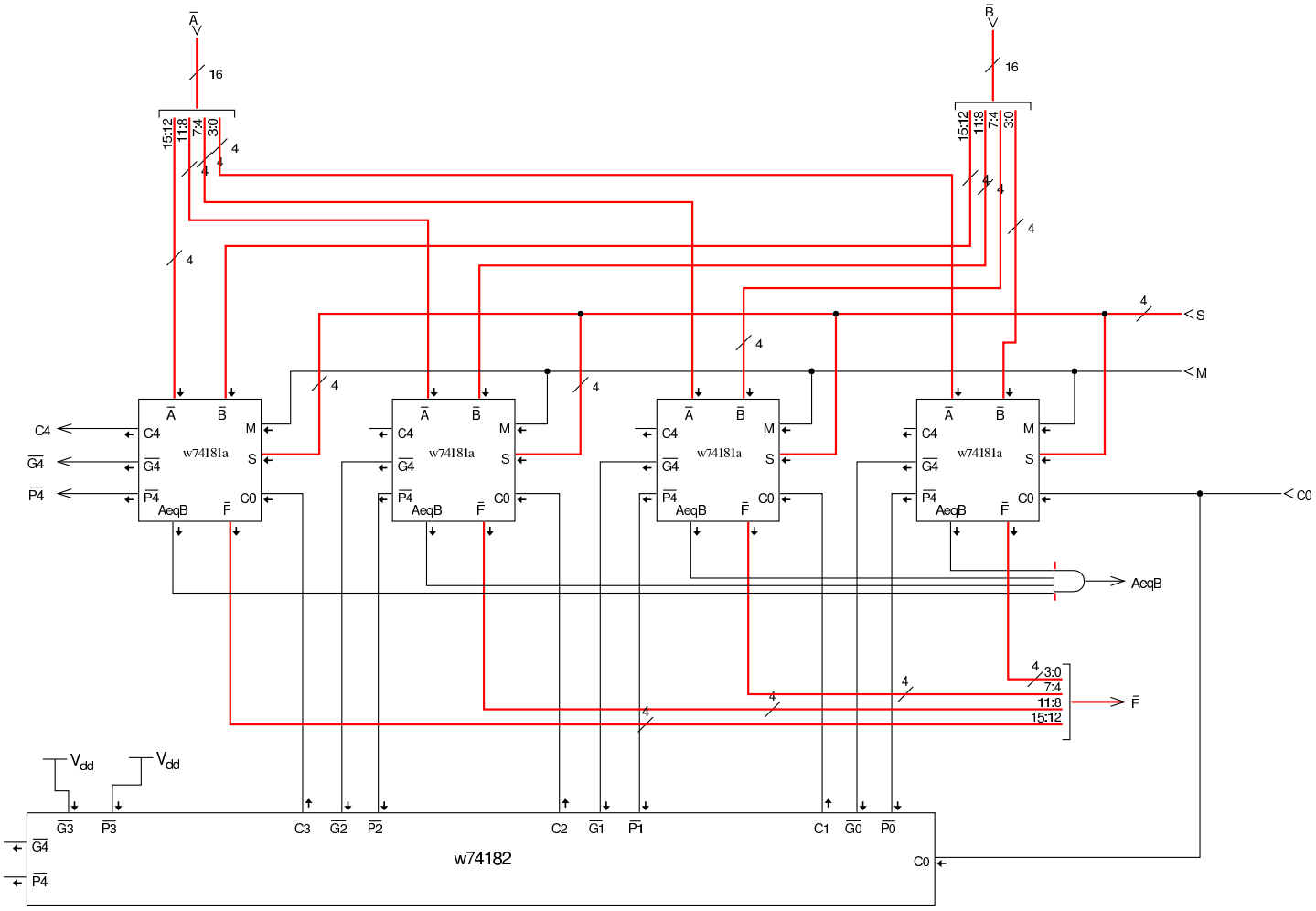

A4
A7
A6
A5
0
1
2
3
4
B4
B6
B7
B5
M
C4
G4
P4
w74181a
S
C0
AeqB
F
S5
S7
S6
S4
AeqB0
F4
F5
F6
F7


S>
4
C0>
M>
B̄ >
4
0
1
2
3
B̄0
Ā0
0
1
2
3
Ā >
4
F0
F̄
AeqB
P̄4
C4
Ḡ4

A 16-bit ALU with four 74181.



A 16-bit ALU with four 74181 and the 74182 as carry lookahead generator.

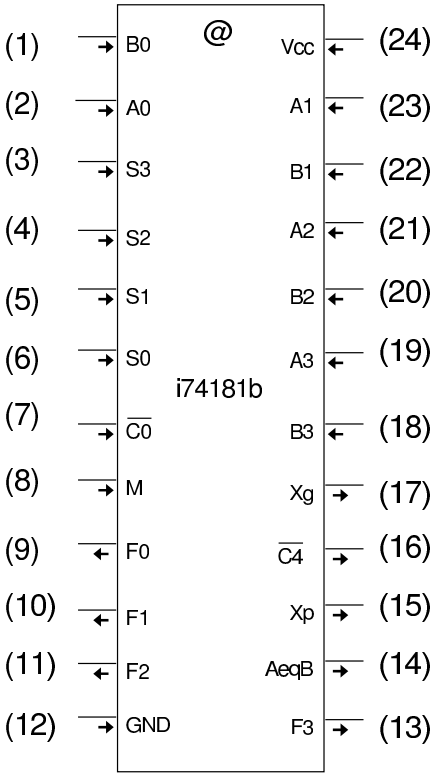


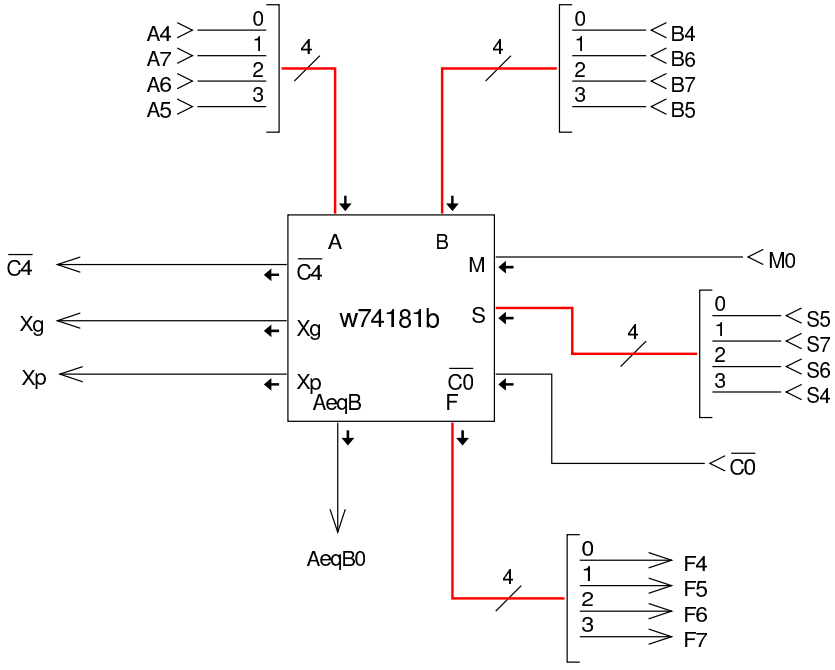
In this mode the 74181 works using the ports ***A***, ***B*** and ***F*** as "active high" whereas the ports ***C0*** and ***C4*** are "active low", but this way the outputs ***G*** and ***P*** do not produce the correct value and should not be used.[2]

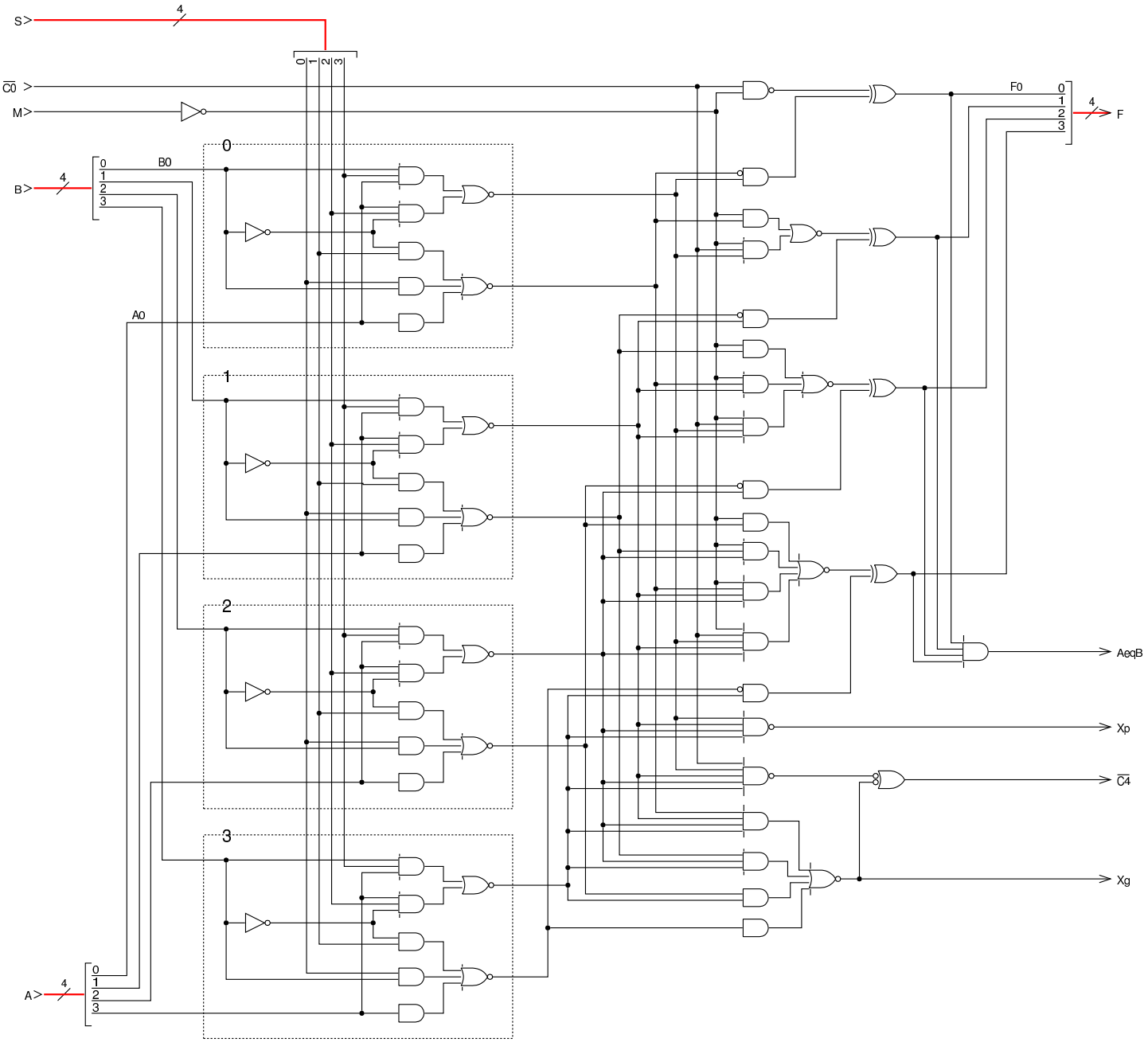
74181 operation table.

$S_{3..0}$	M	$\overline{C}_0$	$\overline{C}_4$	$F_{3..0}$	Xp	Xg
0000	0	0		A plus 1		
0000	0	1		A		
0000	1	X		$\overline{A}$		
0001	0	0		$(A+B)$ plus 1		
0001	0	1		$A+B$		
0001	1	X		$\overline{(A+B)}$		
0010	0	0		$(A+\overline{B})$ plus 1		
0010	0	1		$A+\overline{B}$		
0010	1	X		$\overline{A}\cdot B$		
0011	0	0	0	0000		1
0011	0	1	1	1111		1
0011	1	0	0	0000		1
0011	1	1	1	0000		1
0100	0	0		A plus $(A\cdot\overline{B})$ plus 1		
0100	0	1		A plus $(A\cdot\overline{B})$		
0100	1	X		$\overline{(A\cdot B)}$		
0101	0	0		$A\cdot\overline{B}$ plus $(A+B)$ plus 1		
0101	0	1		$A\cdot\overline{B}$ plus $(A+B)$		
0101	1	X		$\overline{B}$		
0110	0	0		A minus B		
0110	0	1		A minus B minus 1		
0110	1	X		$A\oplus B$		
0111	0	0		$A\cdot\overline{B}$		
0111	0	1		$A\cdot\overline{B}$ minus 1		
0111	1	X		$A\cdot\overline{B}$		
1000	0	0		A plus $(A\cdot B)$ plus 1		
1000	0	1		A plus $(A\cdot B)$		
1000	0	X		$\overline{A}+B$		
1001	0	0		A plus B plus 1		
1001	0	1		A plus B		
1001	1	X		$\overline{(A\oplus B)}$		

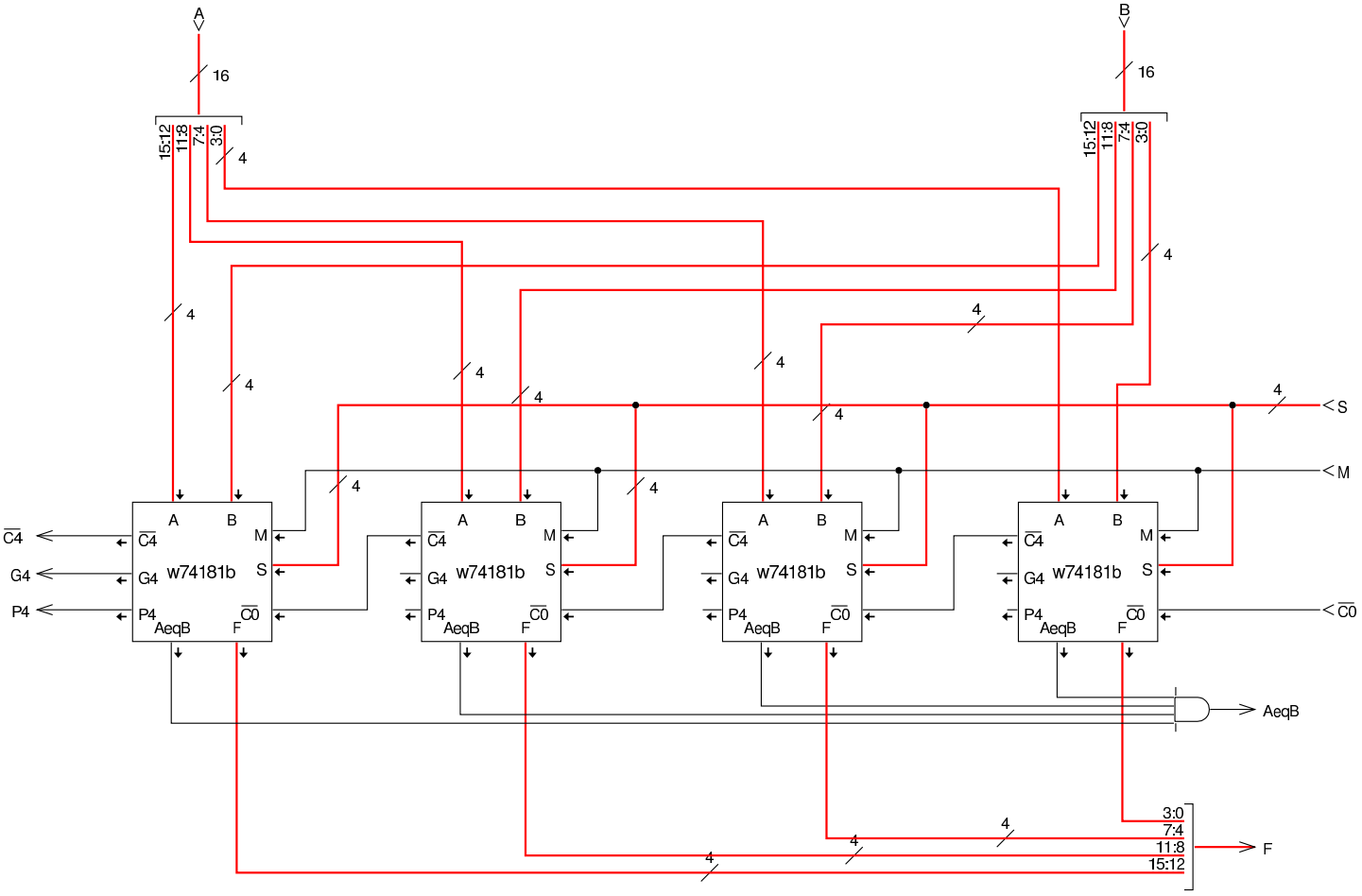
$S_{3..0}$	M	$\overline{C}_0$	$\overline{C}_4$	$F_{3..0}$	Xp	Xg
1010	0	0		$(A \cdot B)$ plus $(A+\overline{B})$ plus 1		
1010	0	1		$(A \cdot B)$ plus $(A+\overline{B})$		
1010	1	X		B		
1011	0	0		$A \cdot B$		
1011	0	1		$A \cdot B$ minus 1		
1011	1	X		$A \cdot B$		
1100	0	0		A plus A plus 1		
1100	0	1		A plus A		
1100	1	X		1111		
1101	0	0		A plus $(A+B)$ plus 1		
1101	0	1		A plus $(A+B)$		
1101	1	X		$A+\overline{B}$		
1110	0	0		A plus $(A+\overline{B})$ plus 1		
1110	0	1		A plus $(A+\overline{B})$		
1110	1	X		$A+B$		
1111	0	0		A		
1111	0	0		A minus 1		
1111	1	X		A		



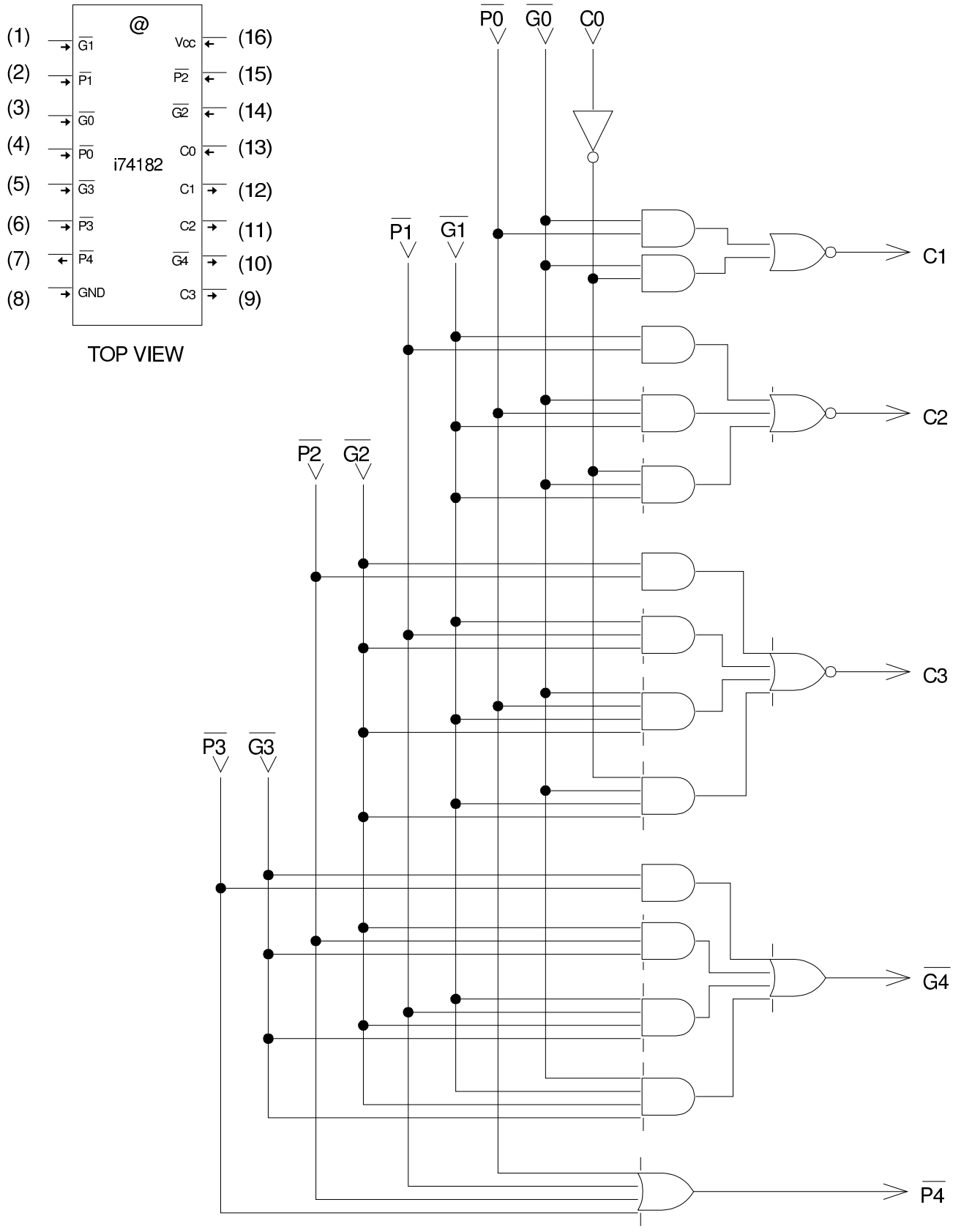



S>
4
C0 >
M>
B>
4
B0
A0
A>
4
0
1
2
3
F0
4
F
AeqB
Xp
C4
Xg

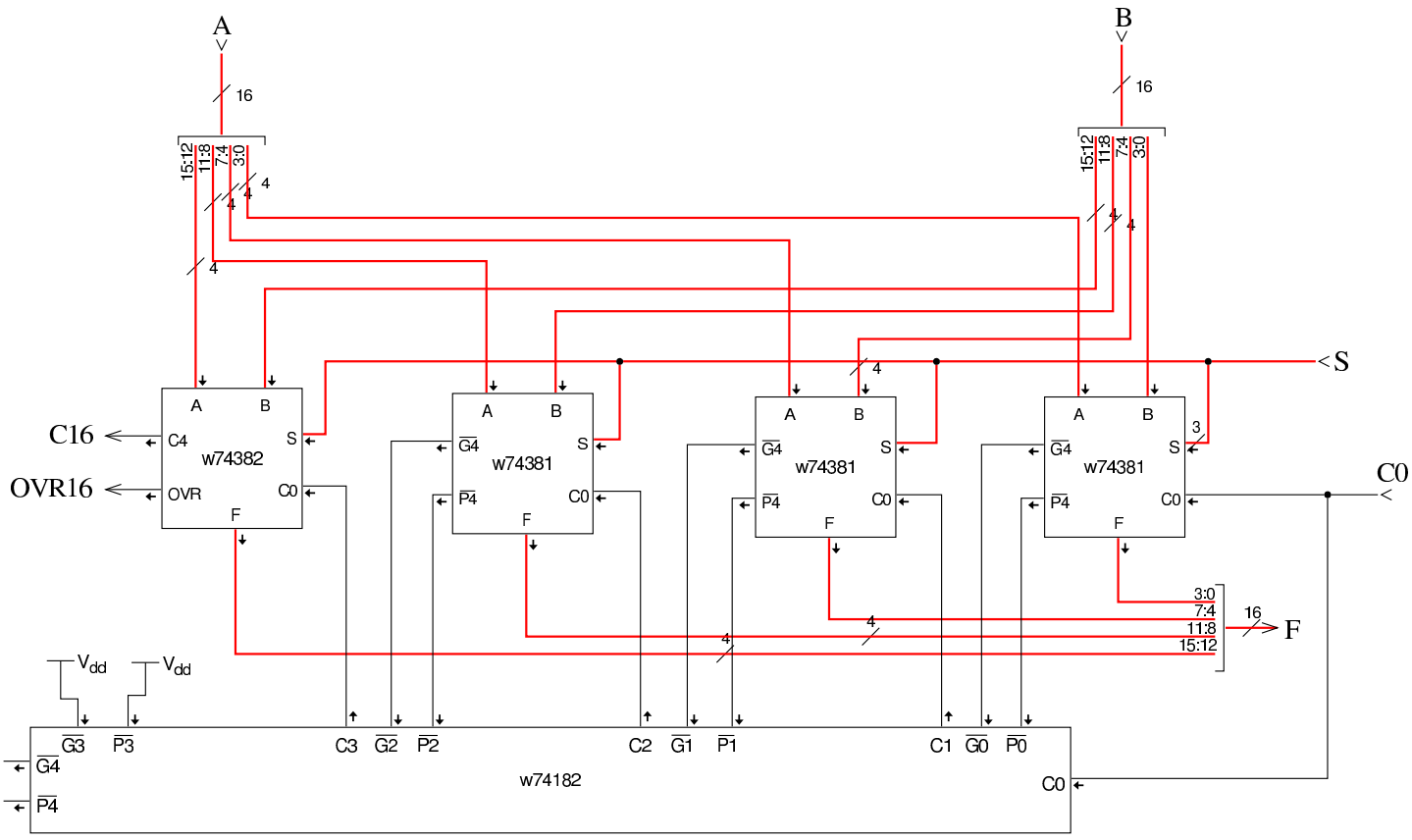
A 16-bit ALU with four 74181: the carry lookahead generator can not be used when input and output data are used as “active high”, because the output ports ***G*** and ***P*** do not work properly.



74182 carry lookahead generator



A 16-bit ALU with 74182 and other ICs.

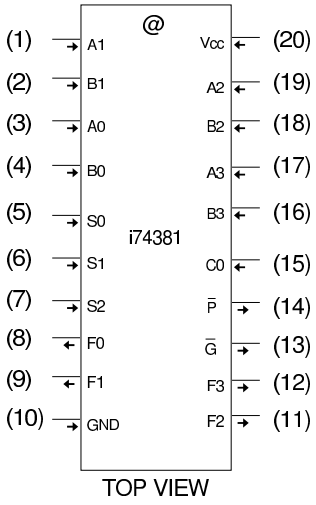


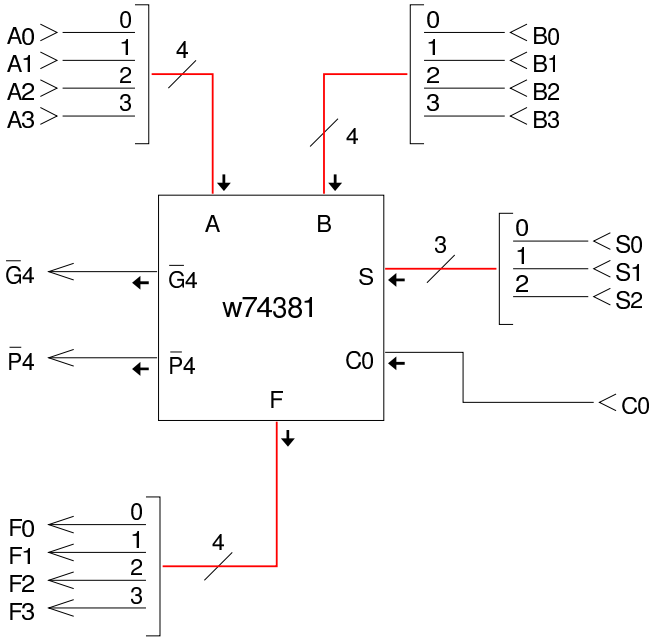
The 74381 is a 4-bit ALU that uses the ‘G’ and ‘P’ values to allow the carry propagation. If a bigger bit-size is required, this component needs the 74182 for the carry lookahead.

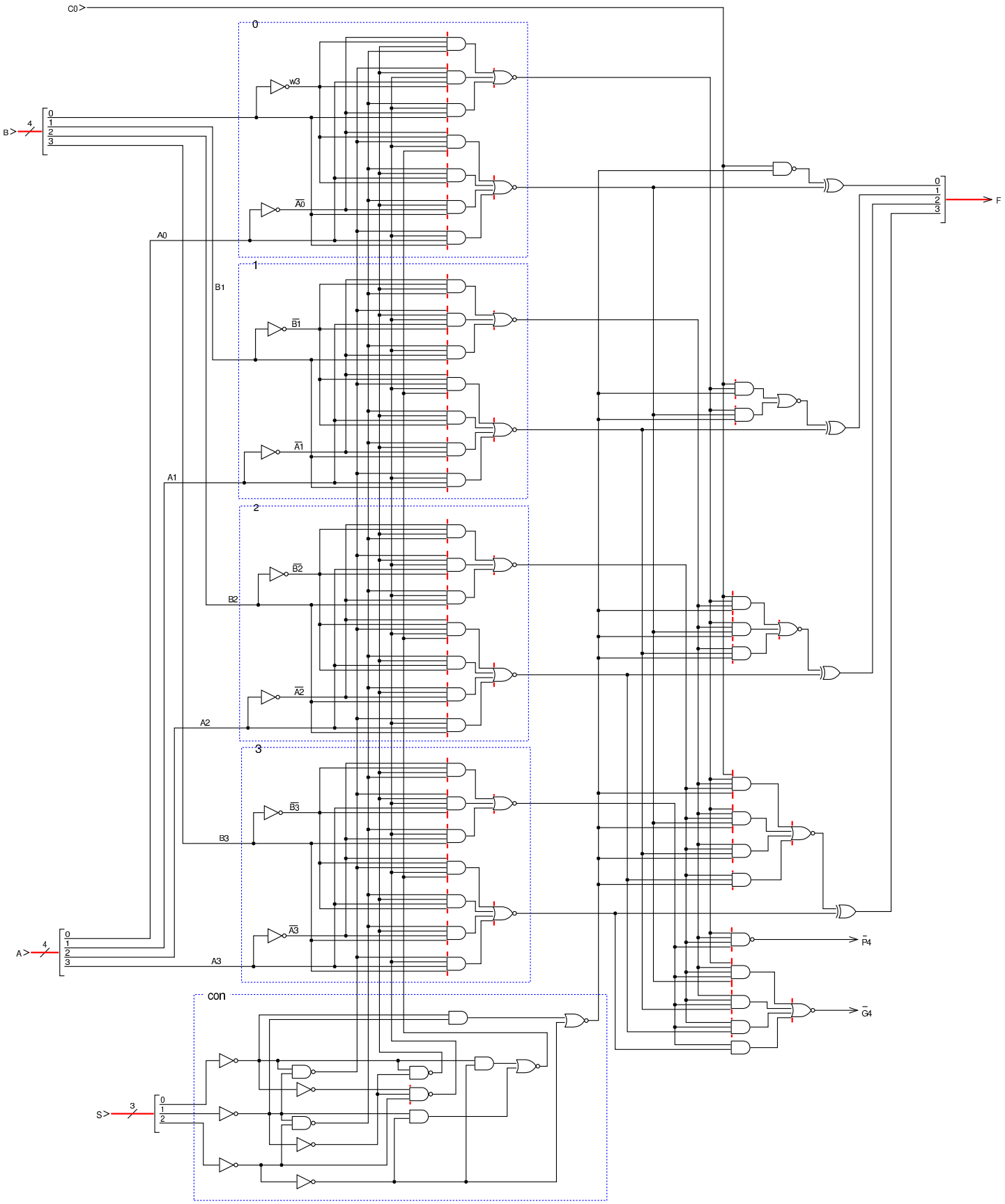
74381 truth table.

Function	$S_{2..0}$	$A_{3..0}$	$B_{3..0}$	C_0	$F_{3..0}$	$\overline{G}_4$	$\overline{P}_4$
clear	000	XXXX	XXXX	X	0000	0	0
B minus ***A***	001	0000	0000	0	1111	1	0
B minus ***A***	001	0000	0000	1	0000	1	0
B minus ***A***	001	0000	1111	0	1110	0	0
B minus ***A***	001	0000	1111	1	1111	0	0
B minus ***A***	001	1111	0000	0	0000	1	1
B minus ***A***	001	1111	0000	1	0001	1	1
B minus ***A***	001	1111	1111	0	1111	1	0
B minus ***A***	001	1111	1111	1	0000	1	0
A minus ***B***	010	0000	0000	0	1111	1	0
A minus ***B***	010	0000	0000	1	0000	1	0
A minus ***B***	010	0000	1111	0	0000	1	1
A minus ***B***	010	0000	1111	1	0001	1	1
A minus ***B***	010	1111	0000	0	1110	0	0
A minus ***B***	010	1111	0000	1	1111	0	0
A minus ***B***	010	1111	1111	0	1111	1	0
A minus ***B***	010	1111	1111	1	0000	1	0
A plus ***B***	011	0000	0000	0	0000	0	0
A plus ***B***	011	0000	0000	1	0001	1	1
A plus ***B***	011	0000	1111	0	1111	1	0
A plus ***B***	011	0000	1111	1	0000	1	0
A plus ***B***	011	1111	0000	0	1111	1	0
A plus ***B***	011	1111	0000	1	0000	1	0
A plus ***B***	011	1111	1111	0	1110	0	0
A plus ***B***	011	1111	1111	1	1111	0	0
$A \oplus B$	100	0000	0000	X	0000	0	0
$A \oplus B$	100	0000	1111	X	1111	1	1
$A \oplus B$	100	1111	0000	X	1111	1	0
$A \oplus B$	100	1111	1111	X	0000	0	0
$A + B$	101	0000	0000	X	0000	0	0
$A + B$	101	0000	1111	X	1111	1	1
$A + B$	101	1111	0000	X	1111	1	1

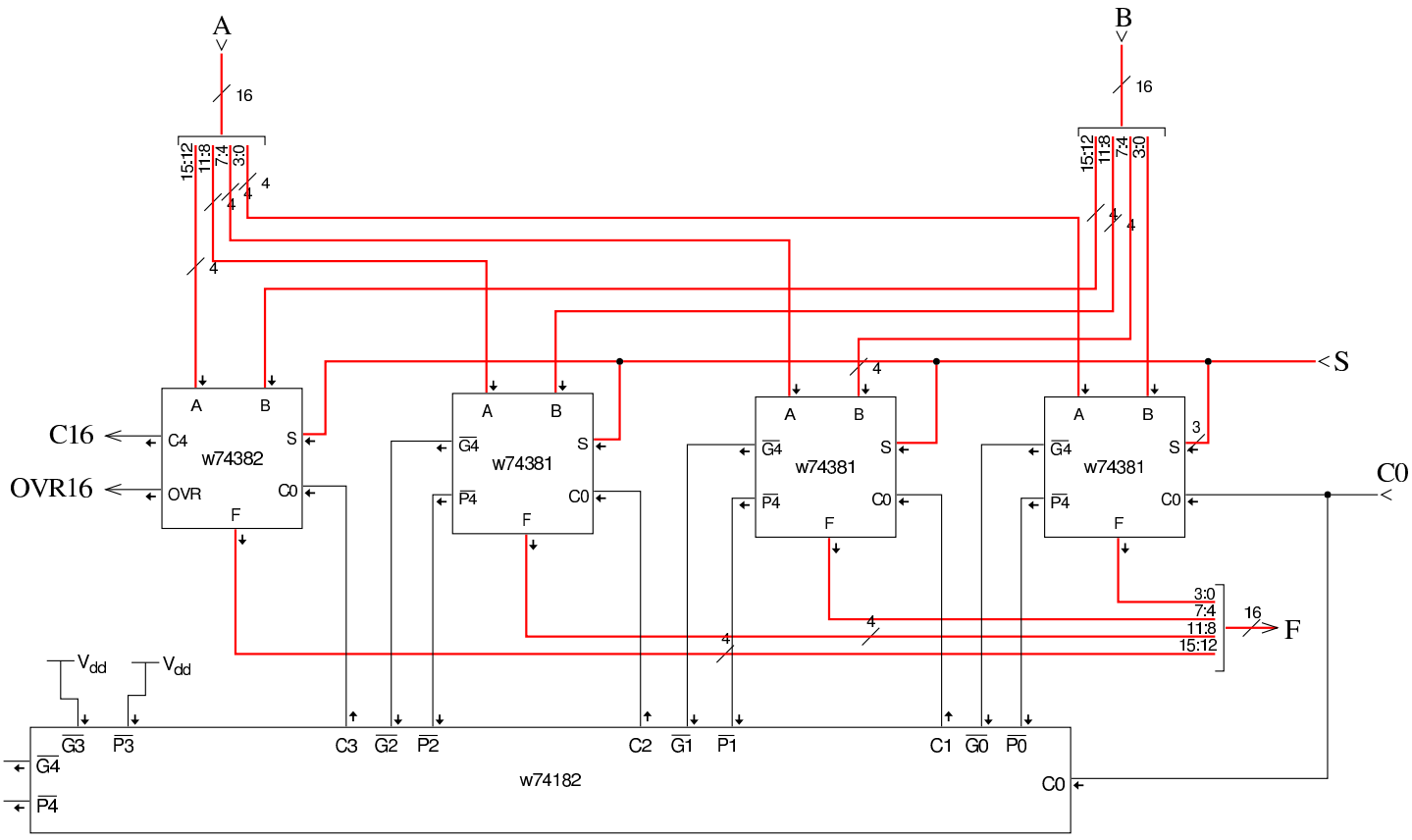
Function	$S_{2..0}$	$A_{3..0}$	$B_{3..0}$	C_0	$F_{3..0}$	$\overline{G}_4$	$\overline{P}_4$
$A + B$	101	1111	1111	X	1111	1	0
$A \cdot B$	110	0000	0000	X	0000	0	0
$A \cdot B$	110	0000	1111	X	0000	1	1
$A \cdot B$	110	1111	0000	X	0000	0	0
$A \cdot B$	110	1111	1111	X	1111	1	0
preset	111	0000	0000	X	1111	1	1
preset	111	0000	1111	X	1111	1	1
preset	111	1111	0000	X	1111	1	1
preset	111	1111	1111	X	1111	1	0






C0
0
w3
A0
1
B1
A1
2
B2
A2
3
B3
A3
B
A
S
con
F
P4
G4

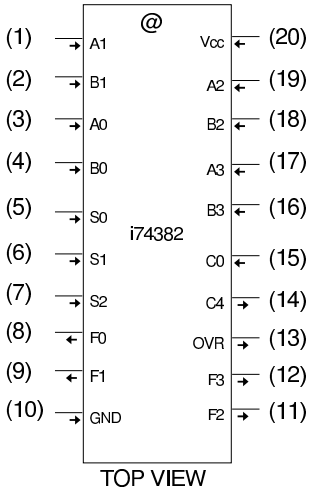
A 16-bit ALU with 74381 and other ICs.

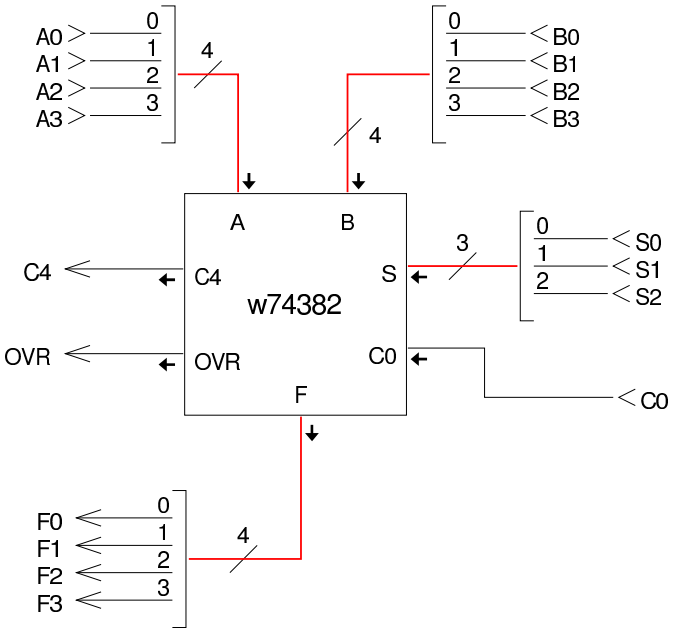


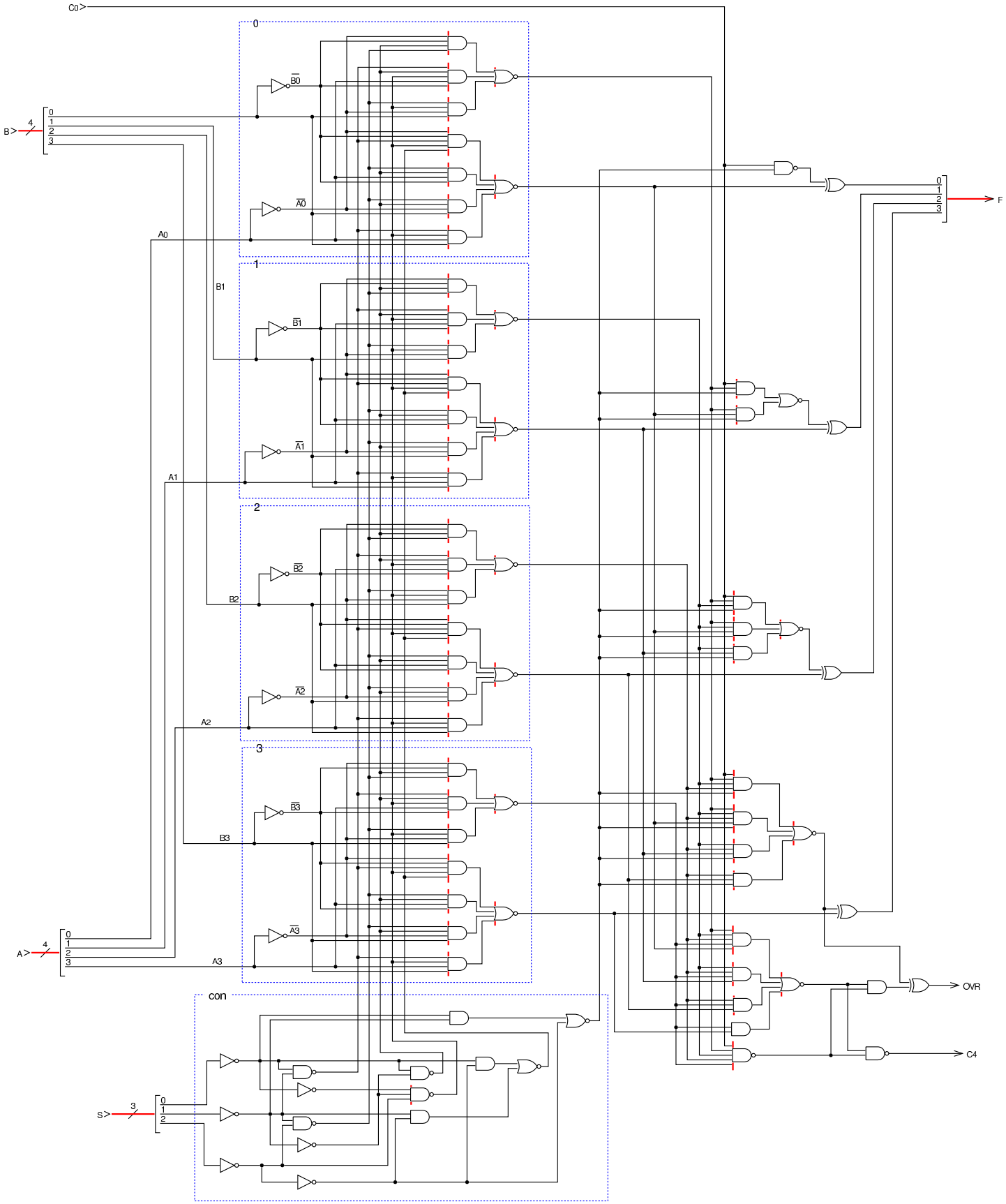
74382 truth table.

Function	$S_{2..0}$	$A_{3..0}$	$B_{3..0}$	C_0	$F_{3..0}$	OVR	C_4
clear	000	XXXX	XXXX	X	0000	1	1
B minus ***A***	001	0000	0000	0	1111	0	0
B minus ***A***	001	0000	0000	1	0000	1	0
B minus ***A***	001	0000	1111	0	1110	0	1
B minus ***A***	001	0000	1111	1	1111	0	1
B minus ***A***	001	1111	0000	0	0000	0	0
B minus ***A***	001	1111	0000	1	0001	0	0
B minus ***A***	001	1111	1111	0	1111	0	0
B minus ***A***	001	1111	1111	1	0000	0	1
A minus ***B***	010	0000	0000	0	1111	0	0
A minus ***B***	010	0000	0000	1	0000	0	0
A minus ***B***	010	0000	1111	0	0000	0	0
A minus ***B***	010	0000	1111	1	0001	0	0
A minus ***B***	010	1111	0000	0	1110	0	1
A minus ***B***	010	1111	0000	1	1111	0	1
A minus ***B***	010	1111	1111	0	1111	0	0
A minus ***B***	010	1111	1111	1	0000	0	1
A plus ***B***	011	0000	0000	0	0000	0	0
A plus ***B***	011	0000	0000	1	0001	0	0
A plus ***B***	011	0000	1111	0	1111	0	0
A plus ***B***	011	0000	1111	1	0000	0	1
A plus ***B***	011	1111	0000	0	1111	0	0
A plus ***B***	011	1111	0000	1	0000	0	1
A plus ***B***	011	1111	1111	0	1110	0	1
A plus ***B***	011	1111	1111	1	1111	0	1
$A \oplus B$	100	0000	0000	X	0000	0	0
$A \oplus B$	100	0000	1111	X	1111	0	0
$A \oplus B$	100	1111	0000	0	1111	0	0
$A \oplus B$	100	1111	0000	1	1111	1	1
$A \oplus B$	100	1111	1111	X	0000	1	1
$A + B$	101	0000	0000	X	0000	0	0
$A + B$	101	0000	1111	X	1111	0	0
$A + B$	101	1111	0000	X	1111	0	0
$A + B$	101	1111	1111	0	1111	0	0
$A + B$	101	1111	1111	1	1111	1	1
$A \cdot B$	110	0000	0000	X	0000	1	1

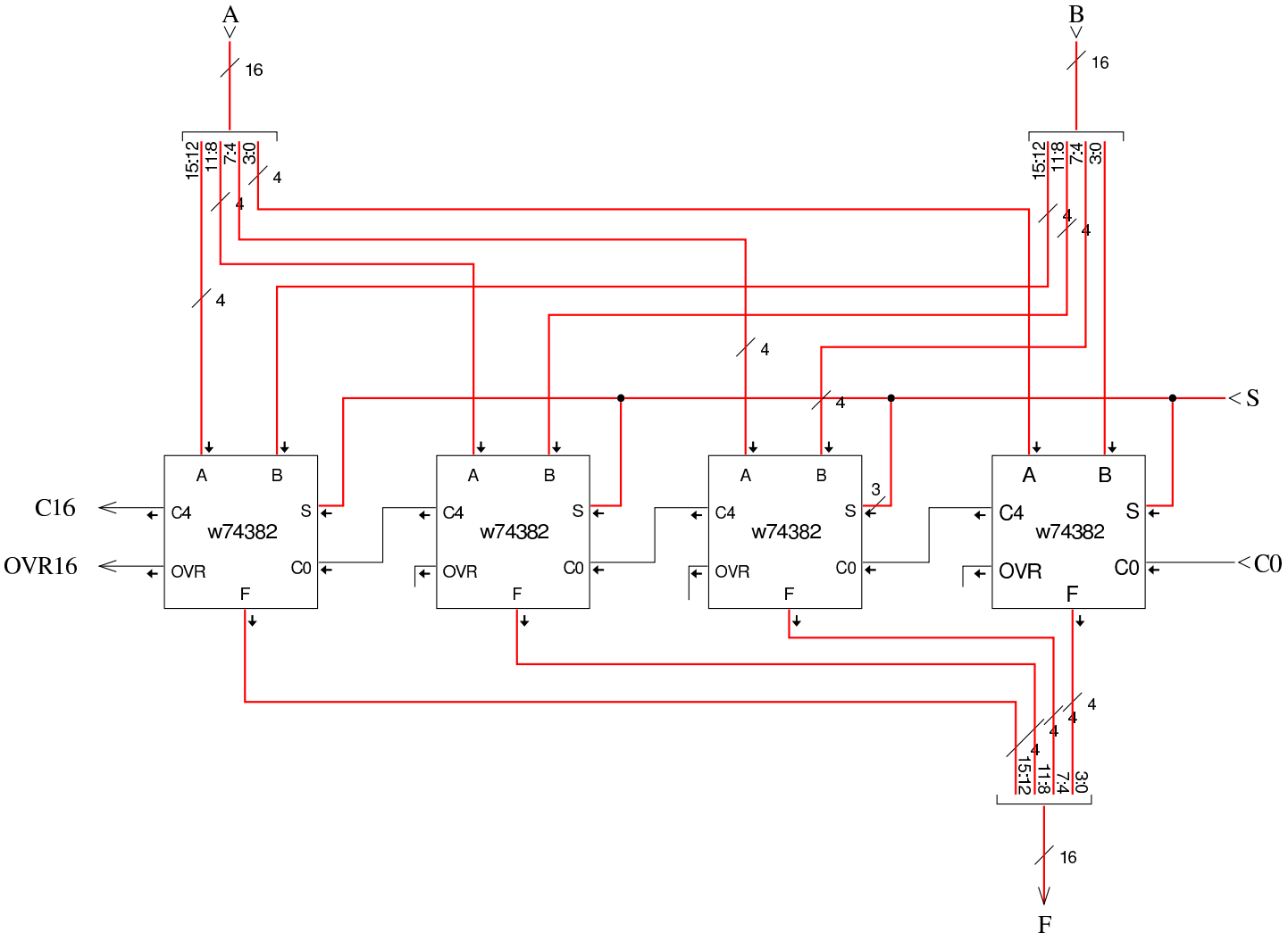
Function	$S_{2..0}$	$A_{3..0}$	$B_{3..0}$	C_0	$F_{3..0}$	OVR	C_4
$A \cdot B$	110	0000	1111	X	0000	0	0
$A \cdot B$	110	1111	0000	X	0000	1	1
$A \cdot B$	110	1111	1111	0	1111	0	0
$A \cdot B$	110	1111	1111	1	1111	1	1
preset	111	0000	0000	X	1111	0	0
preset	111	0000	1111	X	1111	0	0
preset	111	1111	0000	X	1111	0	0
preset	111	1111	1111	0	1111	0	0
preset	111	1111	1111	1	1111	1	1






C0>
0
1
2
3
B0
A0
B>
4
A0
B1
A1
B2
A2
B3
A3
A>
S>
3
con
F
OVR
C4

A 16-bit ALU with four 74382.



[1] The input port ***B*** might be used as “active high” and the operation table should be modified reversing all the ***B*** values: the ***G*** and ***P*** output would work correctly.

[2] The input port ***B*** might be used as “active low” and the operation table should be modified reversing all the ***B*** values, but the ***G*** and ***P*** output would not work either.

References

- Wikipedia, *George Boole*, *http://en.wikipedia.org/wiki/George_Boole*
- Wikipedia, *Augustus De Morgan*, *http://en.wikipedia.org/wiki/Augustus_De_Morgan*
- Wikipedia, *Maurice Karnaugh*, *http://en.wikipedia.org/wiki/Maurice_Karnaugh*
- Albert Paul Malvino, Jerard A. Brown, *Digital Computer Electronics*, Glencoe/Mcgraw-Hill *http://www.amazon.it/Digital-Computer-Electronics-Albert-Malvino/dp/0028005945*
- Stephen Brown, Zvonko Vranesic, *Fundamentals of Digital Logic with Verilog Design*, Mcgraw-Hill *http://www.amazon.com/Fundamentals-Mcgraw-Hill-Electrical-Computer-Engineering/dp/0072823151* ; *Flip-Flops, Registers, Counters, and a Simple Processor*, *http://highered.mcgraw-hill.com/sites/dl/free/0072823151/56549/vra23151_ch07.pdf*
- Olivier Carton, *Circuits et architecture des ordinateurs*, *http://www.liafa.jussieu.fr/~carton/Enseignement/Architecture/archi.pdf*
- Mark Gordon Arnold, *Verilog Digital Computer Design: Algorithms to Hardware*, Prentice Hall PTR *http://www.amazon.com/Verilog-Digital-Computer-Design-Algorithms/dp/0136392539* ; *Verilog hardware description language*, *http://www.pearsonhighered.com/samplechapter/0136392539.pdf*
- IEEE, *IEEE standard verilog hardware description language*, *http://www.google.com/search?q=IEEE+standard+verilog+hardware+description+language*

Part ii

Simple CPU

8 A simple 8-bit CPU: version “A” 307

8.1 Version “A1”: fetch and execution 309
8.2 Version “A2”: memory index 337
8.3 Version “A3”: general purpose registers 348
8.4 Version “A4”: ALU 353
8.5 Version “A5”: flags 384
8.6 Version “A6”: branching 399
8.7 Version “A7”: stack 410
8.8 Version “A8”: I/O 427

9 Version “B”: optimization 445

9.1 Uniform registers 446
9.2 Module “RAM” 449
9.3 Module “SEL” 453
9.4 Module “ALU” 453
9.5 Module “TTY”: the terminal 454
9.6 Module “CTRL”: control unit 459
9.7 Memories, fields, arguments and opcodes 463
9.8 Microcode 479
9.9 Macrocode: routine call 496
9.10 Macrocode: keyboard input and screen output 497
9.11 Version “B2” 498

10 Version “C”: 16-bit little-endian 503

10.1 General purpose 16-bit registers 505
10.2 Module “BUS” 508
10.3 Module “ALU” 509

“from the ground up” draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

10.4 Module “SEL” 518
10.5 Module “RAM” 519
10.6 Module “IRQ” 521
10.7 Module “IVT” 529
10.8 Module “CTRL” 531
10.9 Opcodes 534
10.10 Microcode 558
10.11 Interrupts management 578
10.12 Module “RTC”: real time clock 580
10.13 Module “TTY” 581
10.14 Module “HDD” 586
10.15 Macrocode: terminal usage example, through interrupts 592

11 Version “D” 595

11.1 General purpose registers 598
11.2 Module “VR***n***” 602
11.3 Module “RAM” 605
11.4 Module “ALU” and related submodules 608
11.5 Module “BUS” 615
11.6 Module “IRQ” 616
11.7 Modules “D***n***” 617
11.8 Module “IVT” 618
11.9 Module “RTC”: real time clock 619
11.10 Module “TTY” 621
11.11 Module “HDD” 625
11.12 Module “CTRL” 630
11.13 Memory and microcode fields 636
11.14 Opcodes 641
11.15 Microcode 671
11.16 Macrocode 728

12 Version “E” 731
12.1 General purpose modules 734
12.2 Module “MEM” 739
12.3 General purpose registers 744
12.4 Module “ALU” and related submodules 748
12.5 Module “BUS” 755
12.6 Module “IRQ” 756
12.7 Module “IVT” 757
12.8 Module “RTC”: real time clock 758
12.9 Module “TTY” 759
12.10 Module “HDD” 763
12.11 Module “CTRL” 768
12.12 Memory and microcode fields 772
12.13 Opcodes 776
12.14 Microcode 806
12.15 Macrocode 845
13 Version “F”: 32-bit registers, big-endian, privileges 847
13.1 General purpose modules 850
13.2 Addition modules 853
13.3 Multiple clock lines 857
13.4 Bus 860
13.5 Registers: simple and with post-increment 862
13.6 Memory segmentation 863
13.7 Module “DA“ 864
13.8 Module “CA“ 865
13.9 Modules “M“ and “RAM“ 866
13.10 Module “IR“ 871
13.11 Module “MD“ 873
13.12 Module “IRQ“ 875

13.13 Module “IVT“ 877
13.14 Module “ALU“ 878
13.15 Module “FL“ 887
13.16 Module “TTY” 890
13.17 Module “HDD” 894
13.18 Module “CTRL“ 899
13.19 Memory and microcode fields 902
13.20 Opcodes 904
13.21 Data initialization 934
13.22 Microcode 935
13.23 Process mode and interrupts 966
13.24 Macrocode example: ‘memcpy’ 967
13.25 Macrocode example: TTY with interrupt 969
13.26 Macrocode example: user and supervisor 972
References 981

8.1 Version "A1": fetch and execution 309
8.2 Version "A2": memory index 337
8.2.1 Loading instructions 342
8.2.2 Storing instructions 344
8.3 Version "A3": general purpose registers 348
8.4 Version "A4": ALU 353
8.4.1 Instruction "not" 366
8.4.2 Instruction "and" 368
8.4.3 Instruction "or" 370
8.4.4 Instruction "xor" 372
8.4.5 Instruction "lshl" and "lshr" 374
8.4.6 Instruction "ashl" and "ashr" 376
8.4.7 Instruction "rotl" and "rotr" 378
8.4.8 Instruction "add" 380
8.4.9 Instruction "sub" 382
8.5 Version "A5": flags 384
8.5.1 Instruction "rotcl" and "rotcr" 390
8.5.2 Instruction "add_carry" 392
8.5.3 Instruction "sub_borrow" 396
8.6 Version "A6": branching 399
8.7 Version "A7": stack 410
8.7.1 Instruction "push" and "pop" 422
8.7.2 Instruction "call" and "return" 424
8.8 Version "A8": I/O 427
8.8.1 Communication with asynchronous devices 427
8.8.2 I/O devices implementation 429

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

8.8.3 External appearance of the synchronous interfaces 434
8.8.4 Synchronous keyboard interface 436
8.8.5 Synchronous display interface 437
8.8.6 The CPU data bus with the I/O interfaces 438
8.8.7 Instruction "out" 443
8.8.8 Instruction "in" 444

fetch 309 macrocode 309 microcode 309 microcode word 309 microinstruction 309 opcode 309 operation code 309

Here is introduced the development of a simple CPU, gradually adding components and functions, up to a very simple computer. Initially, it is only a CPU with 8-bit registers, including those relating to addressing of the RAM memory, which is thus limited to a maximum of 256 bytes.

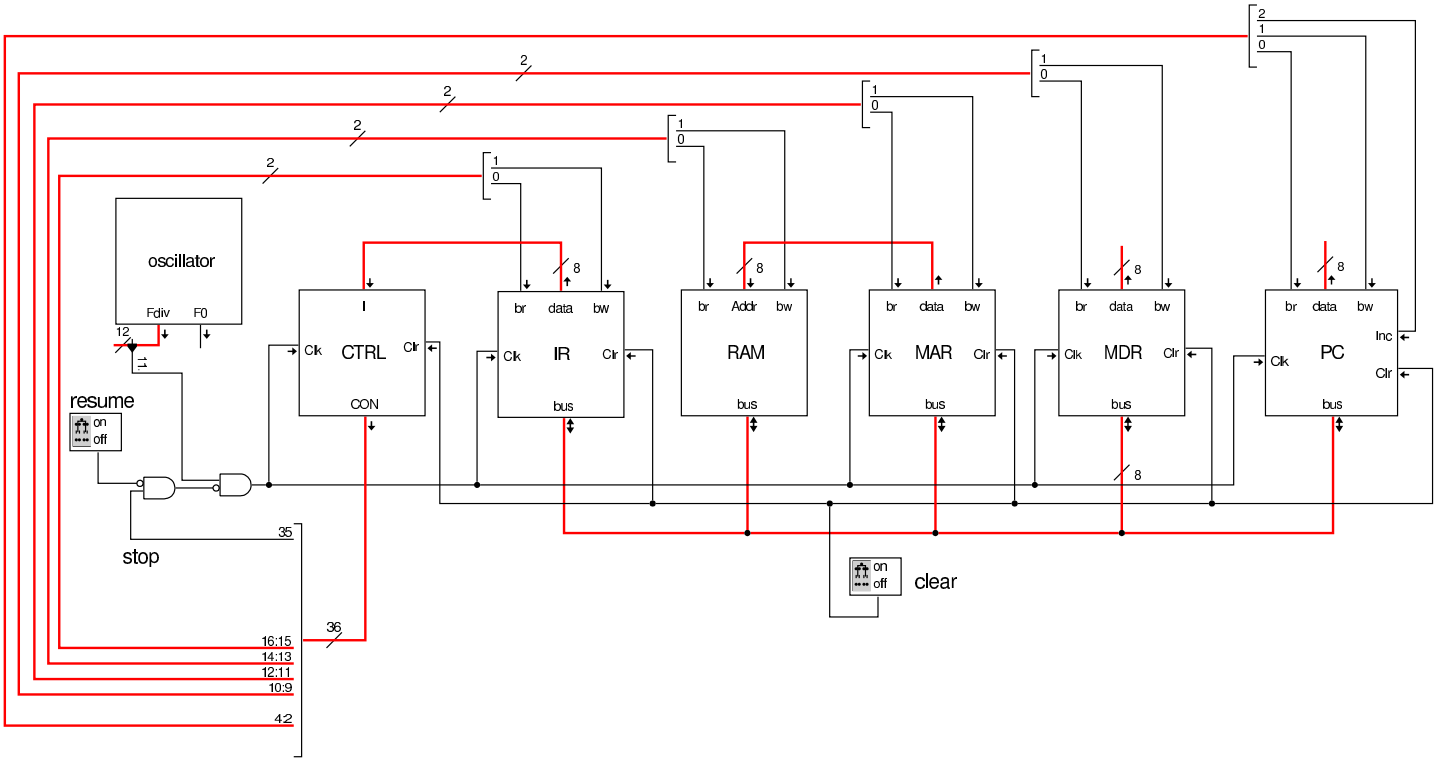
Attachments	Description
attachments/xcpu/xcpu-a1.v	TKGate Verilog netlist source file for the CPU version "A1".
attachments/xcpu/xcpu-a1.gm	TKGate microcode and macrocode source file for the CPU version "A1".
attachments/xcpu/xcpu-a2.v	TKGate Verilog netlist source file for the CPU version "A2".
attachments/xcpu/xcpu-a2.gm	TKGate microcode and macrocode source file for the CPU version "A2".
attachments/xcpu/xcpu-a3.v	TKGate Verilog netlist source file for the CPU version "A3".
attachments/xcpu/xcpu-a3.gm	TKGate microcode and macrocode source file for the CPU version "A3".
attachments/xcpu/xcpu-a4.v	TKGate Verilog netlist source file for the CPU version "A4".
attachments/xcpu/xcpu-a4-not.gm *attachments/xcpu/xcpu-a4-and.gm* *attachments/xcpu/xcpu-a4-or.gm* *attachments/xcpu/xcpu-a4-xor.gm* *attachments/xcpu/xcpu-a4-lsh.gm* *attachments/xcpu/xcpu-a4-ash.gm* *attachments/xcpu/xcpu-a4-rot.gm* *attachments/xcpu/xcpu-a4-add.gm* *attachments/xcpu/xcpu-a4-sub.gm*	TKGate microcode and macrocode alternative source files for the CPU version "A4".

Attachments	Description
attachments/xcpu/xcpu-a5.v	TKGate Verilog netlist source file for the CPU version "A5".
attachments/xcpu/xcpu-a5-rotc.gm *attachments/xcpu/xcpu-a5-addc.gm* *attachments/xcpu/xcpu-a5-subb.gm*	TKGate microcode and macrocode alternative source files for the CPU version "A5".
attachments/xcpu/xcpu-a6.v	TKGate Verilog netlist source file for the CPU version "A6".
attachments/xcpu/xcpu-a6.gm	TKGate microcode and macrocode source file for the CPU version "A6".
attachments/xcpu/xcpu-a7.v	TKGate Verilog netlist source file for the CPU version "A7".
attachments/xcpu/xcpu-a7-push-pop.gm *attachments/xcpu/xcpu-a7-call-return.gm*	TKGate microcode and macrocode alternative source files for the CPU version "A7".
attachments/xcpu/xcpu-a8.v	TKGate Verilog netlist source file for the CPU version "A8".
attachments/xcpu/xcpu-a8-kbd.vpd.tcl	TCL/Tk script related to the keyboard module **KBD**.
attachments/xcpu/xcpu-a8-dsp.vpd.tcl	TCL/Tk script related to the screen module **DSP**.
attachments/xcpu/xcpu-a8-out.gm *attachments/xcpu/xcpu-a8-in.gm*	TKGate microcode and macrocode alternative source files for the CPU version "A8".

8.1 Version "A1": fetch and execution

In its first attempt, the CPU consists only of registers useful to access the memory, to read the opcode that should be executed, as shown in the next figure.

Figure 8.2. Simple CPU version "A1".



The simpler module that can be analyzed is the oscillator, which produces the clock signal. It is an oscillator constructed with a series of inverting logic gates, to create a propagation delay sufficient to produce an oscillation at a manageable frequency. To activate the oscillation is required an initial pulse, which, after a short time to zero, is activated permanently. The next figure shows the oscillator and the start pulse necessary for activation. It is important to note that the series of inverter gates must be in odd number, as if it were a single inverter gate, but with a long propagation delay. The result is then given to a frequency divider, composed in this case of a chain of synchronous T flip-flops; the output is made of many lines grouped together, each of which allows to select an oscillation at a different frequency. The frequency divider is initialized by the same initial pulse, which starts from a state to zero. In the case of the examples is used a very low frequency, obtained from the last stage of division.

Figure 8.3. Oscillator used for the clock signal.

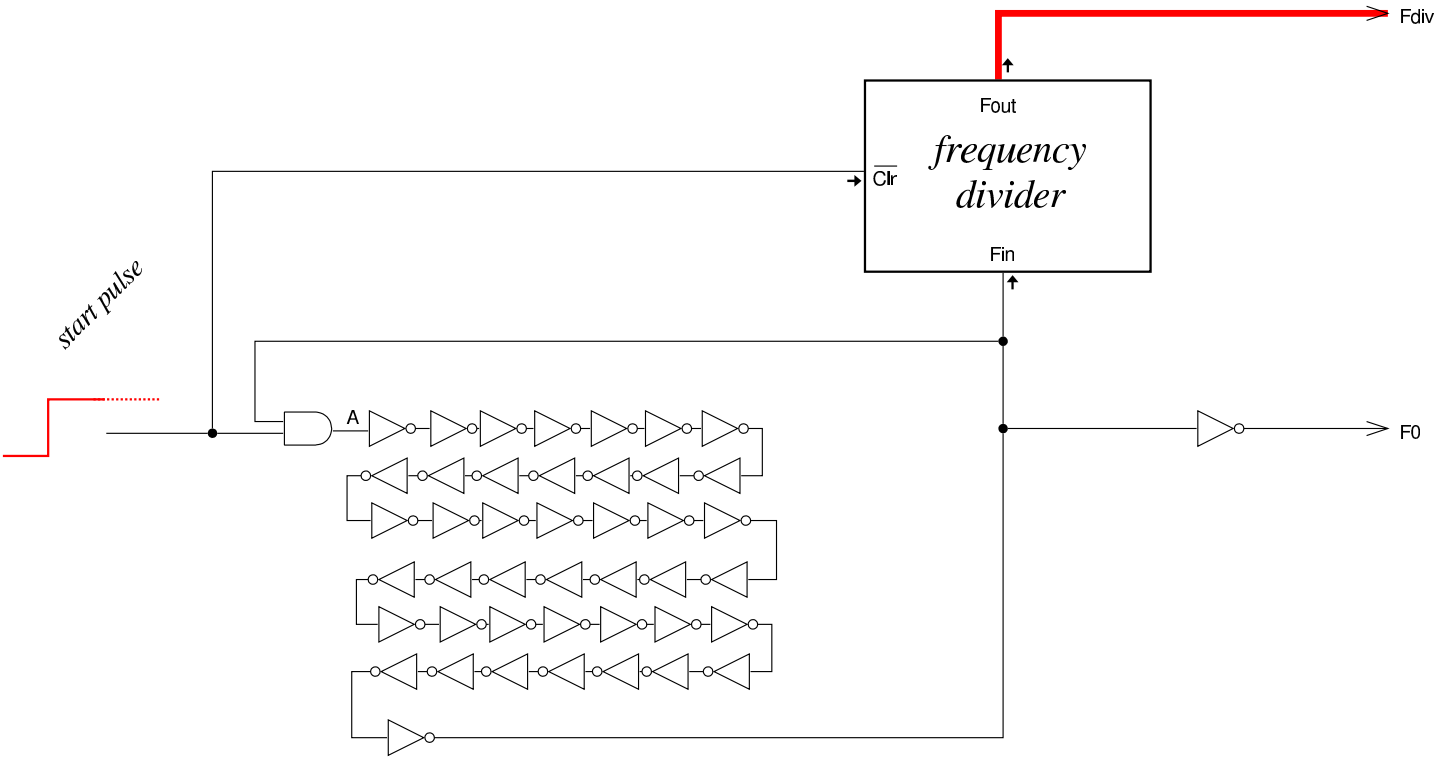
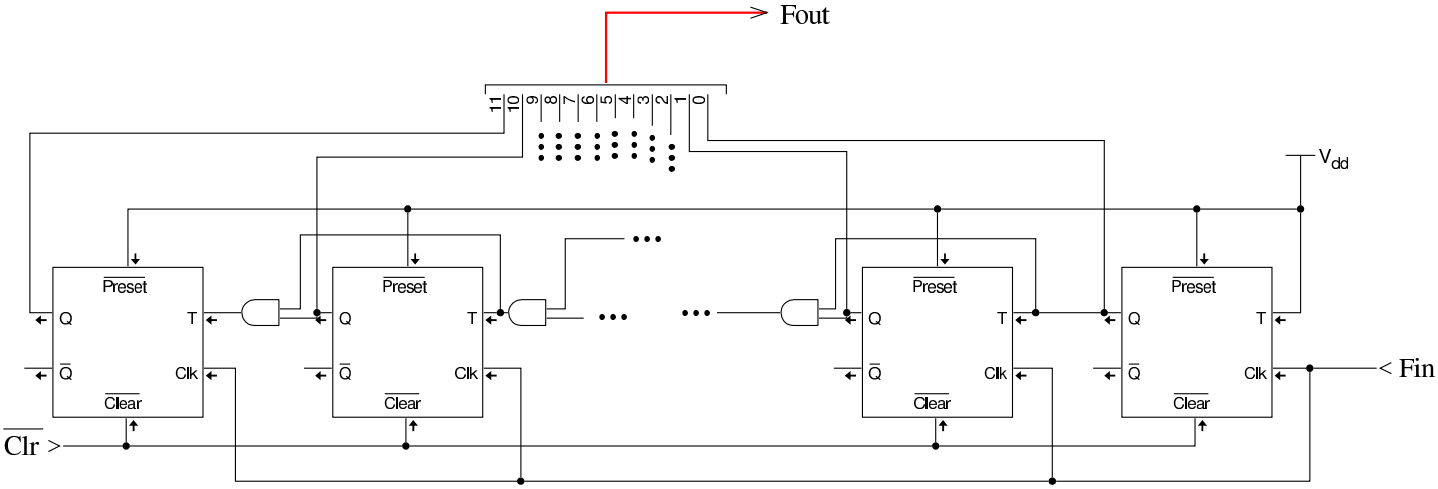


Figure 8.4. Frequency divider used inside the oscillator module: the divider is made of 12 T flip-flop, but the figure shows only four of them.



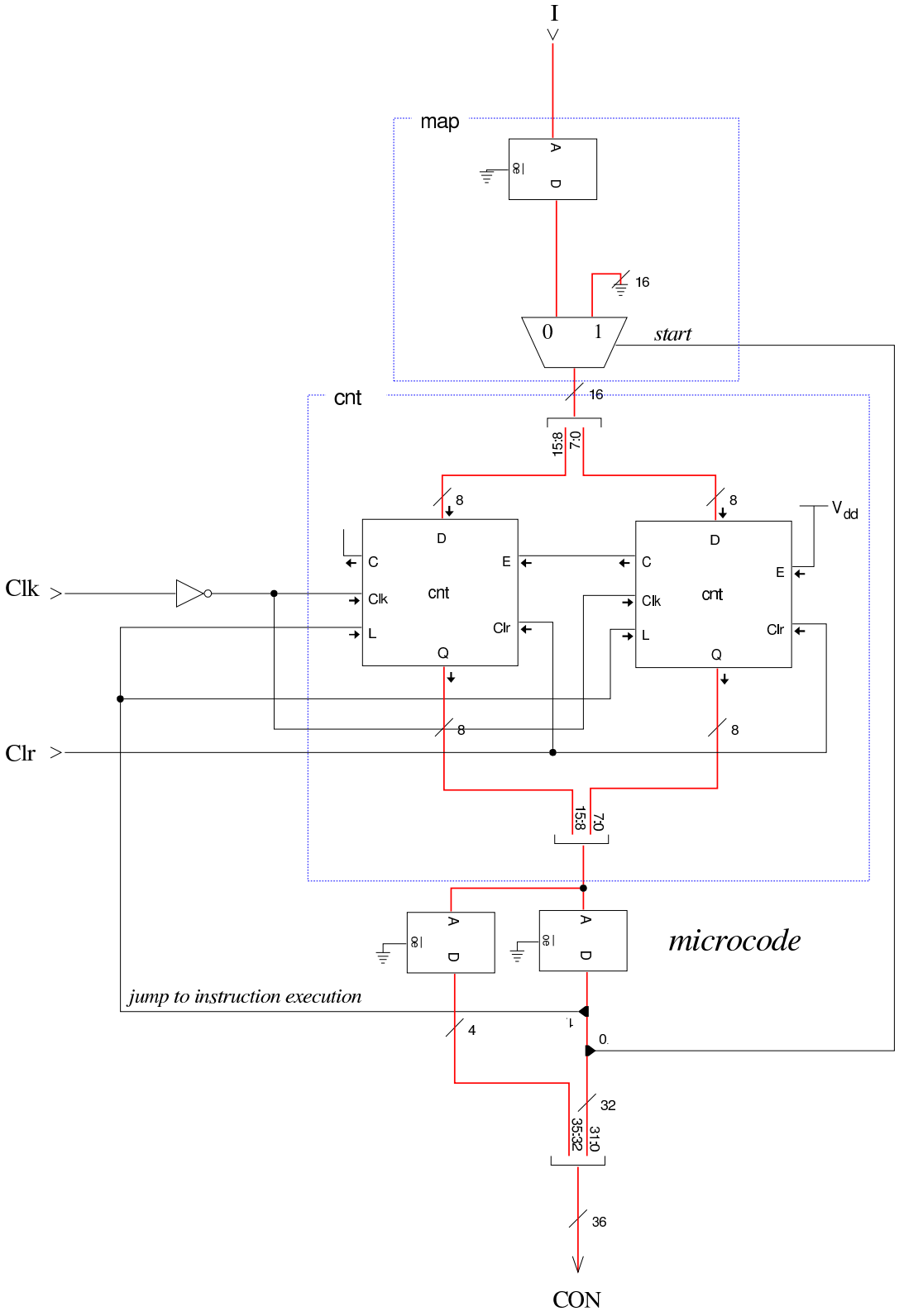
The initial impulse is produced by a component synthesized through the Verilog code because otherwise would be necessary to use analog electronic components and their discussion is beyond the scope of this study.

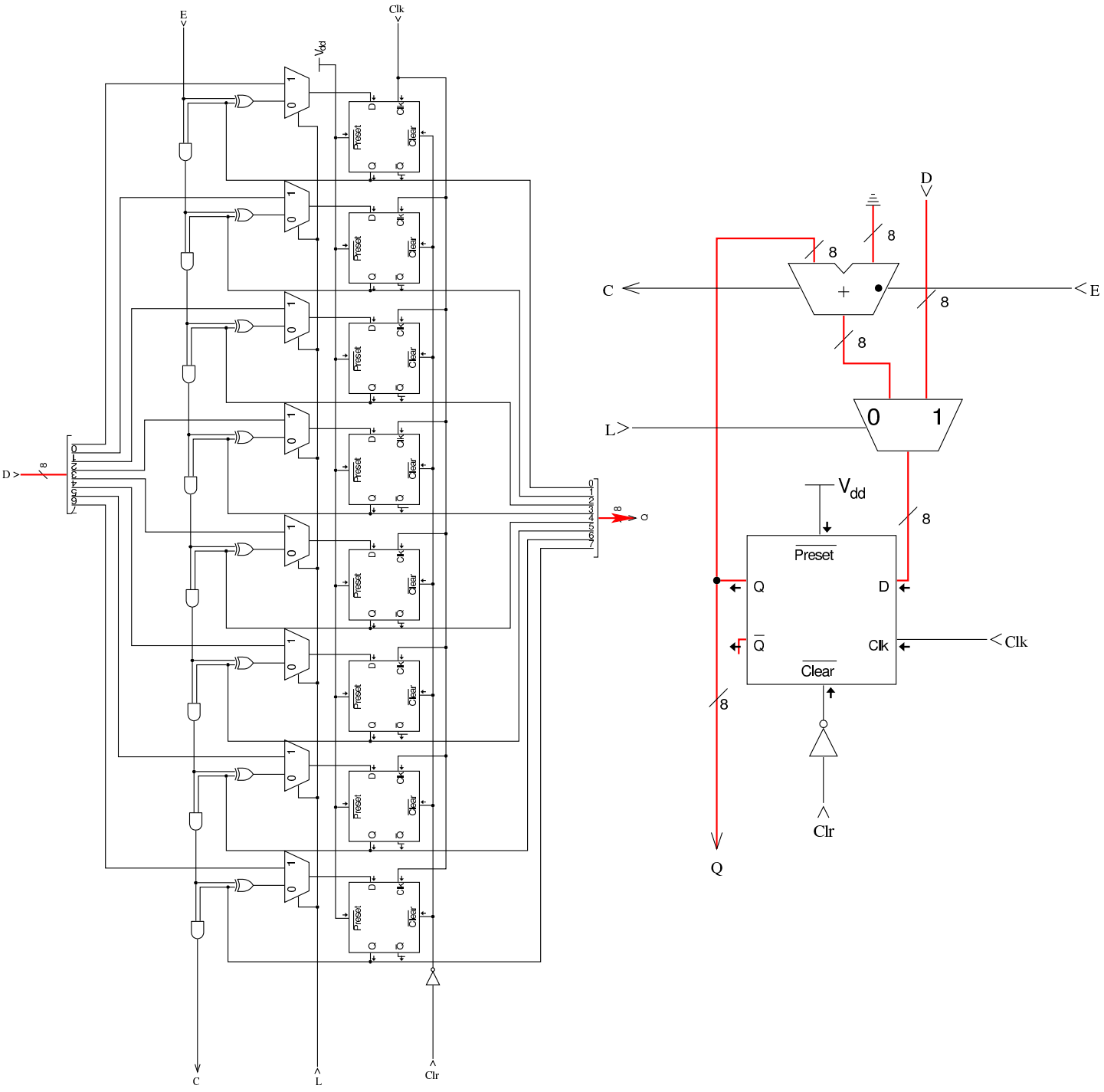
Figure 8.5. Verilog code for TKGate, relative to the trigger module used to start the oscillation: the output is initially zero, and after a brief moment it passes to one, thus remaining permanently.

```
module one_up (Z);
  output Z;
  reg Z;
  initial
    begin
      Z = 1'b0;
      // wait 1 ms: the timescale is 1 ns
      #1000000;
      // activate the output value
      Z = 1'b1;
    end
endmodule
```

The control unit, contained in the **CTRL** module, is very similar to that described in section 7.3, with the difference that the input is identified by the 8-bit variable ***I*** (the letter 'I' stands for 'instruction') and that the output has a much higher bit range, forcing to use two memory units in parallel. The counter that is used to scan the instructions in the final block of memory has a total of 16 bits, but for convenience, two 8-bit counters were used in cascade. The input ***I*** of the control unit is fed by the ***IR*** register content (*instruction register*).

Figure 8.6. Control unit: first there is the 8-bit counter module (microprogram counter) based on D flip-flops; then it appears the detail of the counter, in two alternative implementtions.

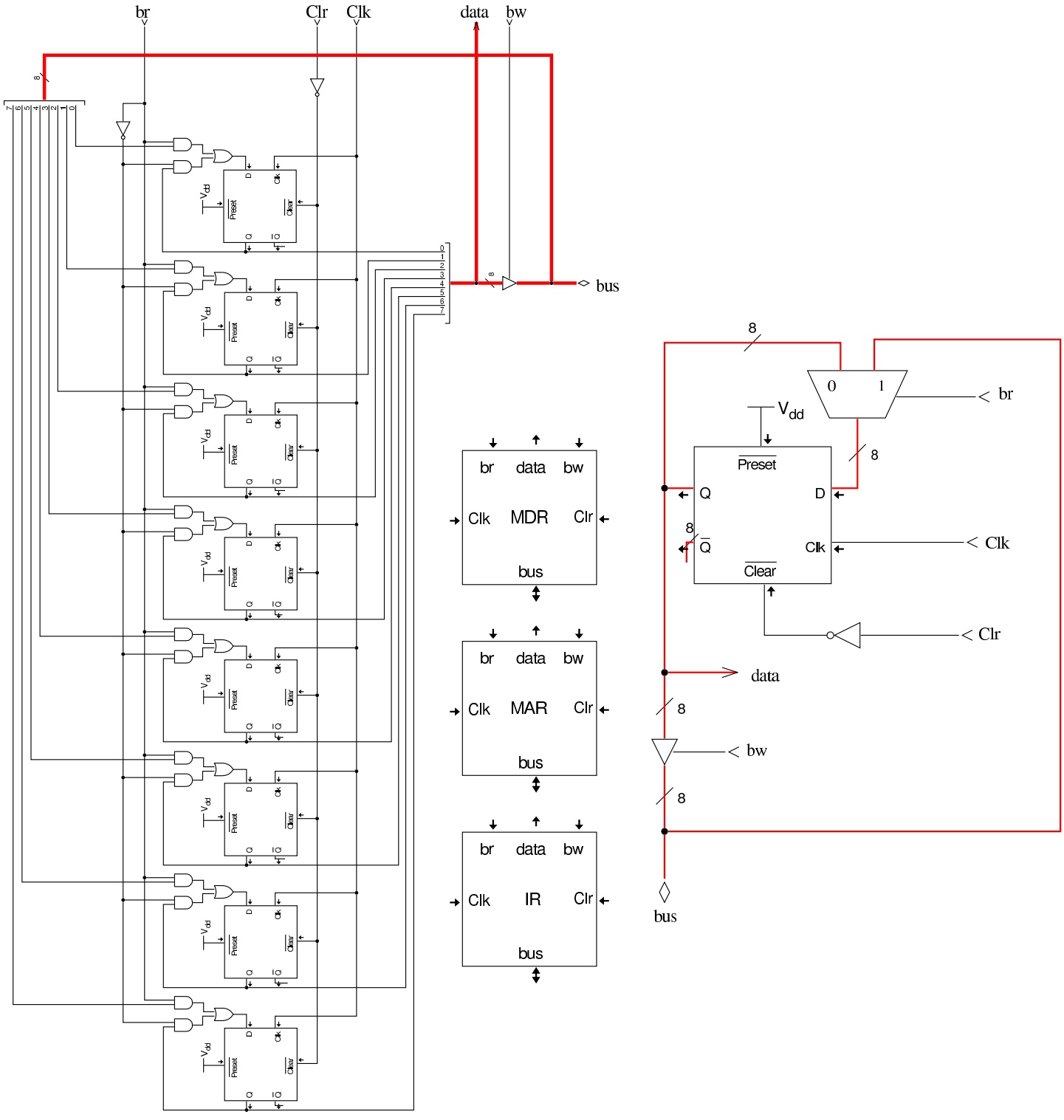




The modules ***IR***, ***MAR*** and ***MDR***, are simple registers, constructed with D flip-flops, connected to the bus through tri-state buffers. From these register it is possible to pick up a copy of the value stored by an additional output, called ***data***. The register ***IR*** (*instruction register*), already mentioned, has the purpose to keep the opcode that the control unit should execute; the register ***MAR*** (*memory address register*) has the purpose to keep the memory address to which the memory is to be accessed; the register ***MDR*** (*memory data register*) is used to accumulate what is read from the

memory for some reason or what is to be written to the memory.

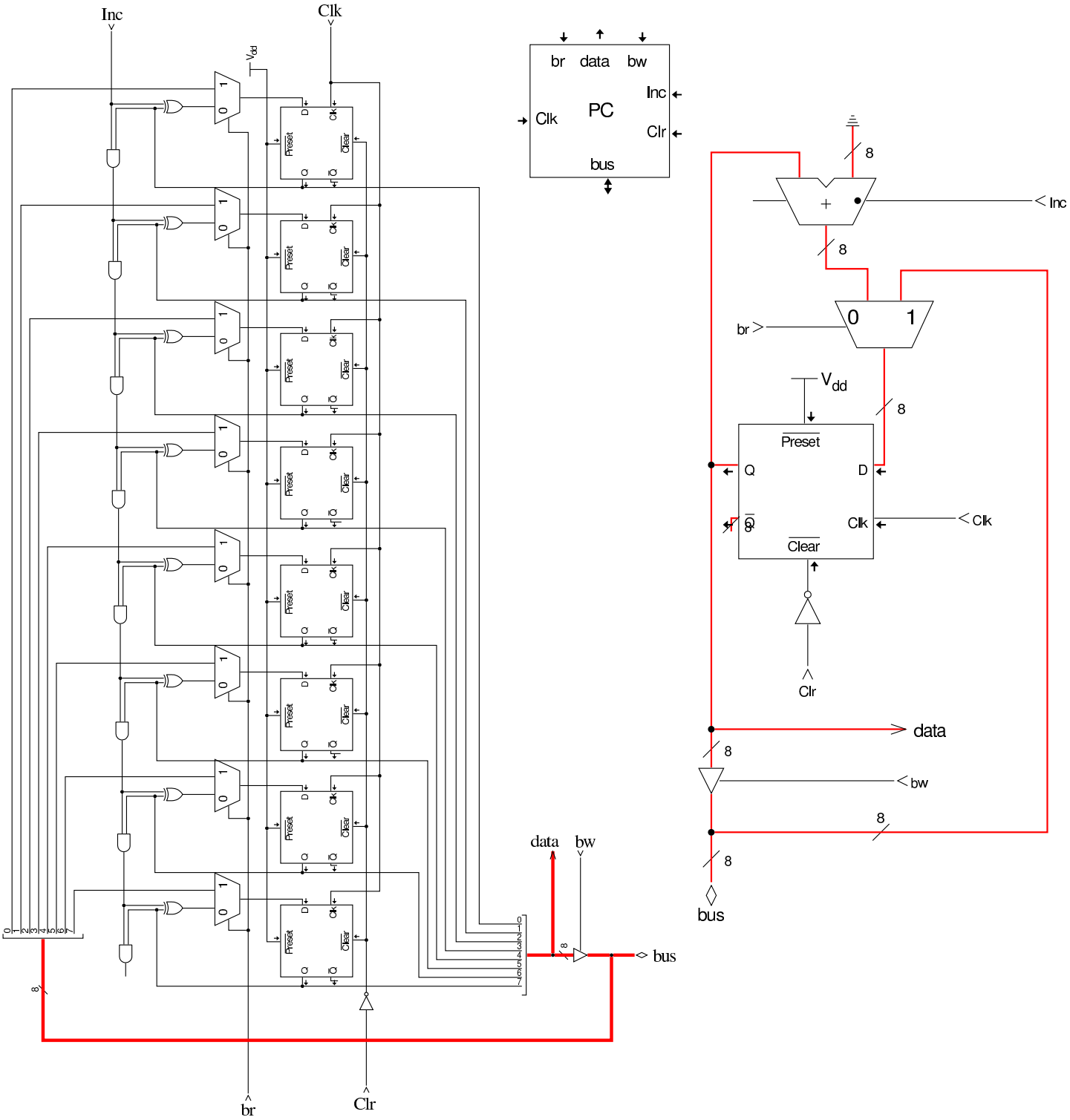
Figure 8.7. Registers ***IR***, ***MAR*** and ***MDR***. The diagram on the right is a compact version of the same circuit.



The ***PC*** module is a register similar to the others, with the difference that it may increase the value that it contains when the input ***Inc*** is asserted. The register ***PC***

(*program counter*) is intended to contain the memory address of the next opcode to be executed.

Figure 8.8. Counter register ***PC***. The diagram on the right is a compact, equivalent circuit.

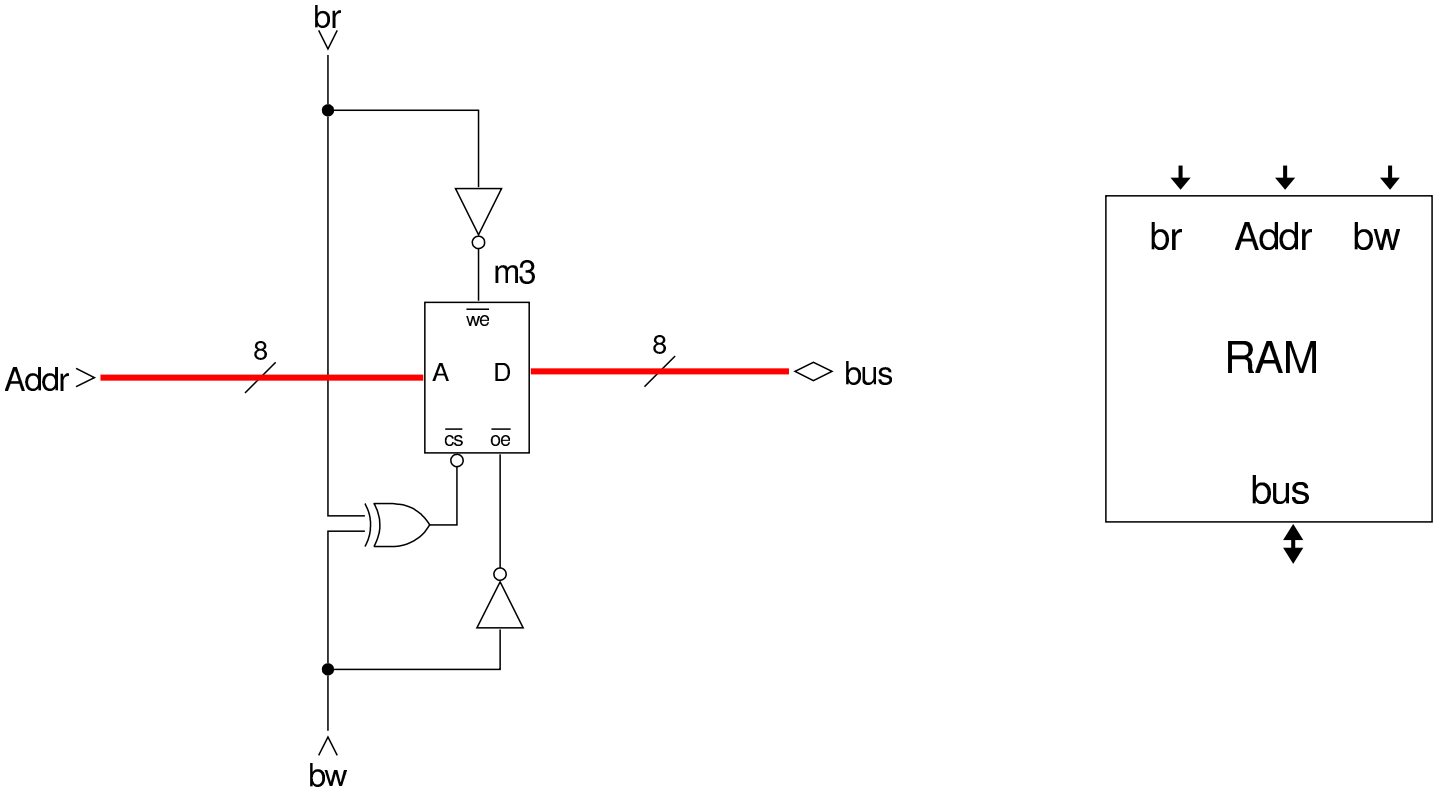


The **RAM** module is substantially different from the others, as it contains the RAM

used by the CPU. This memory is accessed through the address provided via the 8-bit ***Addr*** input, but also the content of the memory is organized in 8-bit cells. The module shares with the others the same inputs to control the data bus access, but when the **RAM** module receives the address and the data bus read is enabled, the value in memory is updated immediately (except for the propagation delay), without waiting until the clock pulse reaches a positive edge.

The **RAM** module receives the address from the ***MAR*** register, which is dedicated to hold the memory address to which the memory access should be done.

Figure 8.9. Module **RAM**. The logic network that controls the inputs ***br*** and ***bw***, prevents the simultaneous read and write of the data bus.



The first thing that the structure described above should do is to load an opcode instruction, followed by the execution of it: this is known as ***fetch cycle***. In this structure the ***PC*** register contains the address of the instruction to be executed: this value must be transferred to the ***MAR*** register and the ***PC*** register is incremented: the instruction is obtained from the RAM at the address specified by the ***MAR*** register and it is copied to the ***IR*** register. This can be summarized as follows:

1. ***MAR*** = ***PC***
2. ***PC***++

3. ***IR = RAM[MAR]***

The following figures show just three steps, pointing out the values of ***br***, ***bw*** and ***Inc*** inputs, with LEDs, which turns red when the line to which they are connected is activated. The figures show the time when the clock signal becomes active.

Figure 8.10. First phase: it is requested to the ***PC*** register to send its value to the data bus and to the ***MAR*** register to read it. This implements in practice the operation ***MAR=PC***.

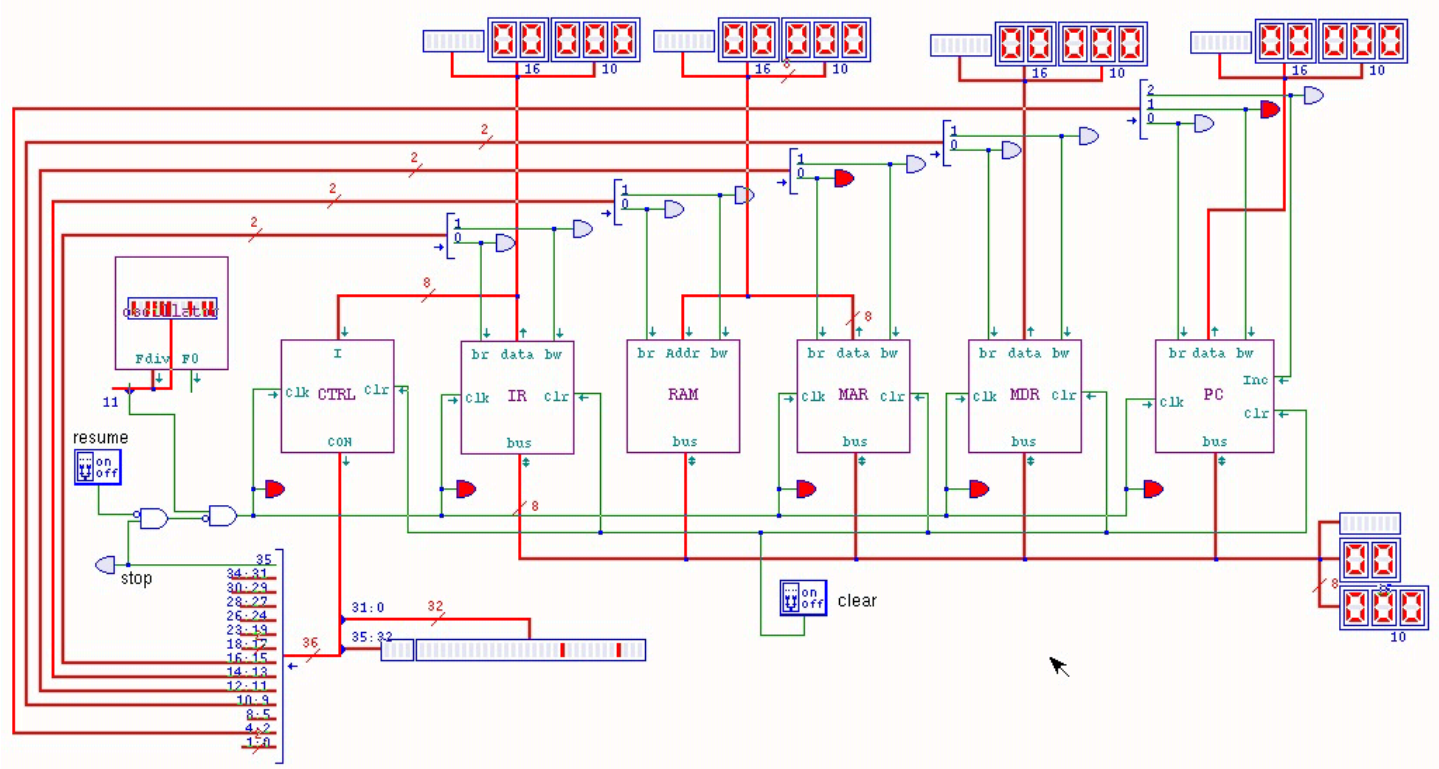


Figure 8.11. Second phase: it is requested to the ***PC*** register to increase by one. This implements in practice the operation ***PC***++.

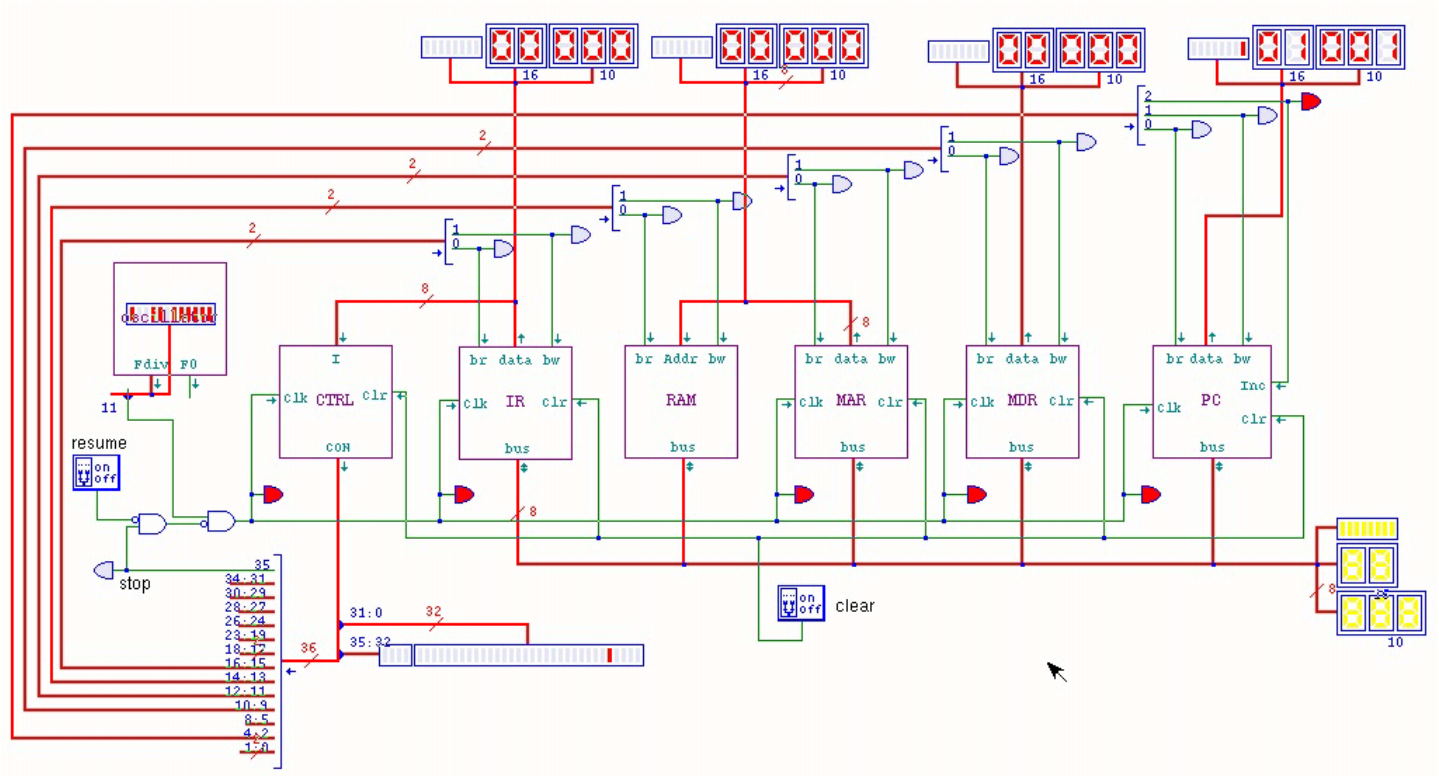
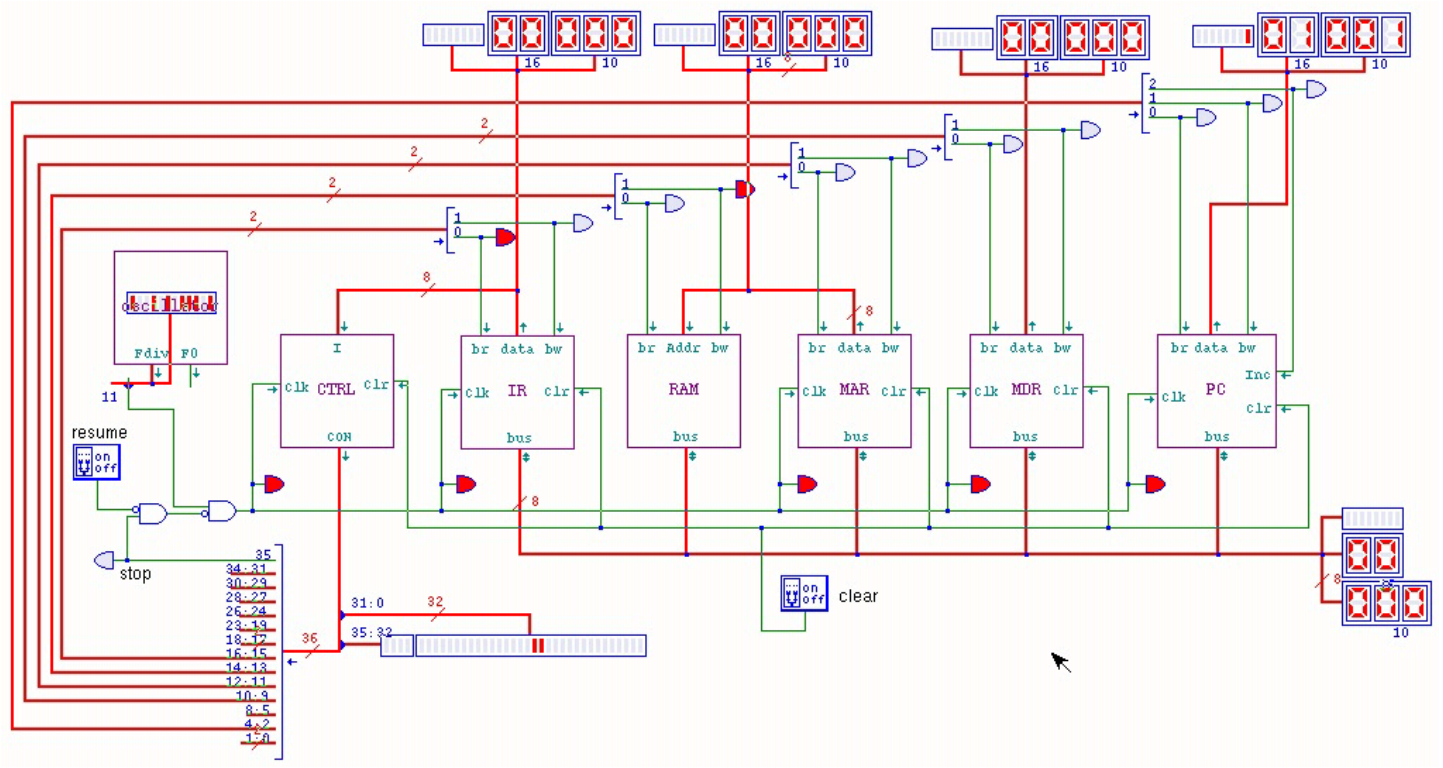


Figure 8.12. Third phase: it is requested to the RAM to send the value at the address pointed by the ***MAR*** register to the data bus and it is requested to the ***IR*** register to load it. This implements in practice the operation ***IR=RAM***[***MAR***].



Within the control unit (the **CTRL** module) time is marked in the same way, apart from the fact that the counters **cnt** are driven by an inverted clock signal, to anticipate the activation of the control lines with respect to the data bus. Initially the counters of the control unit are cleared, so that they point to the first microinstruction, which is the request to perform the operation ***MAR=PC***. Then, the microcode counter (the group of the two counters **cnt**) is incremented and the new value points to the second microinstruction, which is the request to increase the ***PC*** register. Then, there is a further counter increase, bringing to the microinstruction ***IR=RAM***[***MAR***].

Figure 8.13. First phase: the counters of the control unit are reset and the initial microinstruction is ***MAR=PC***.

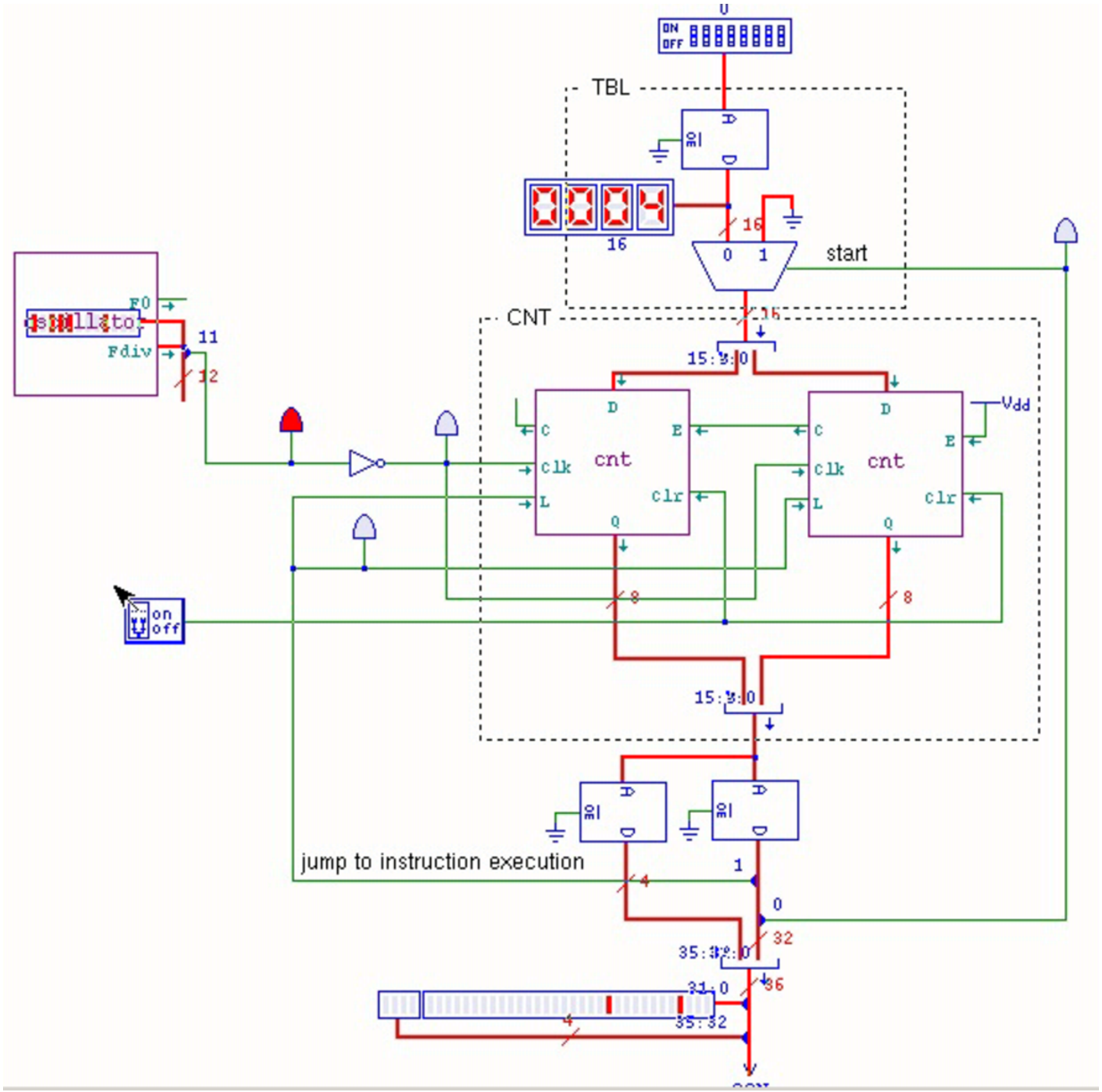


Figure 8.14. Second phase: the set of counters has been increased and the microinstruction produced is ***PC***++.

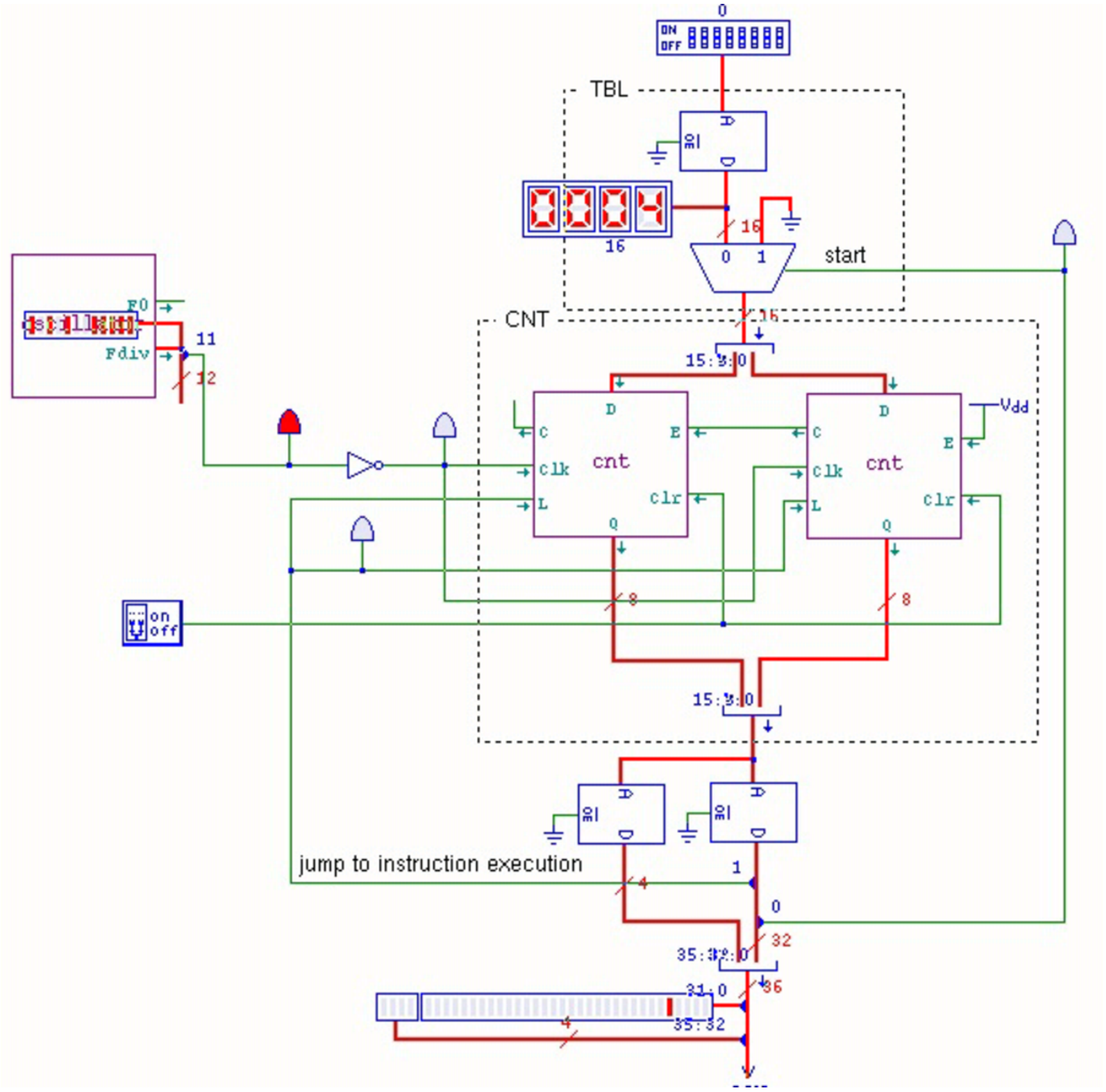
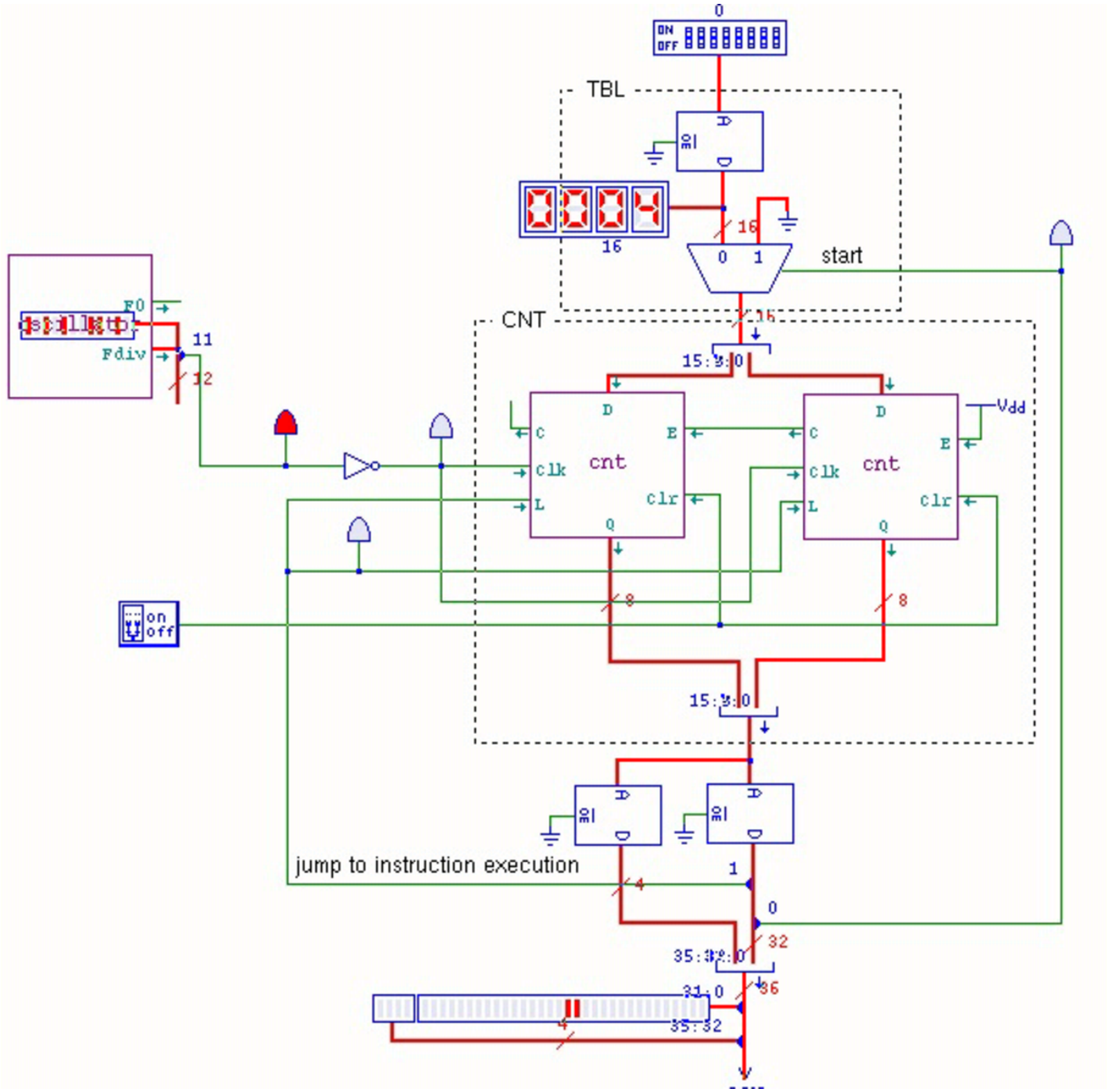


Figure 8.15. Third phase: the set of counters has been increased and the microinstruction produced is ***IR=RAM*[*MAR*]**.



At this point, the control unit has received the opcode from the input ***I*** and is ready to execute it. To do this, the next microinstruction requires the microcode counter to accept the input value. This value is the content of the memory bank **`m0`** at the address represented by the opcode itself. The value obtained from the memory bank **`m0`** is the microcode address to be executed. In the examples of the figures, the opcode is

00000000_2 that is named **`not_operate`** (do nothing).

Figure 8.16. Fourth phase: the counters of the control unit are loaded with the value from the memory bank that translates the opcode instruction into a microcode address.

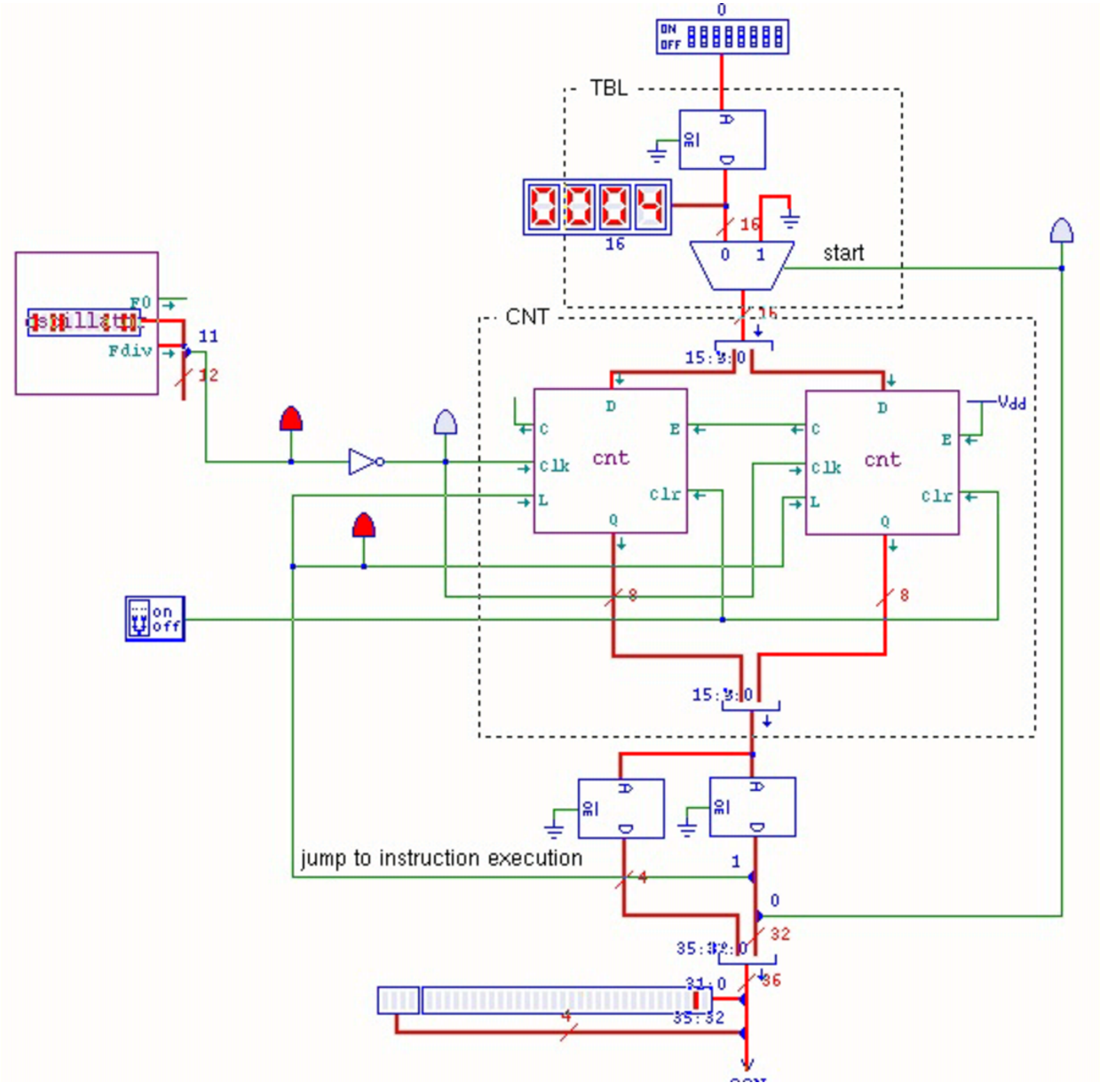
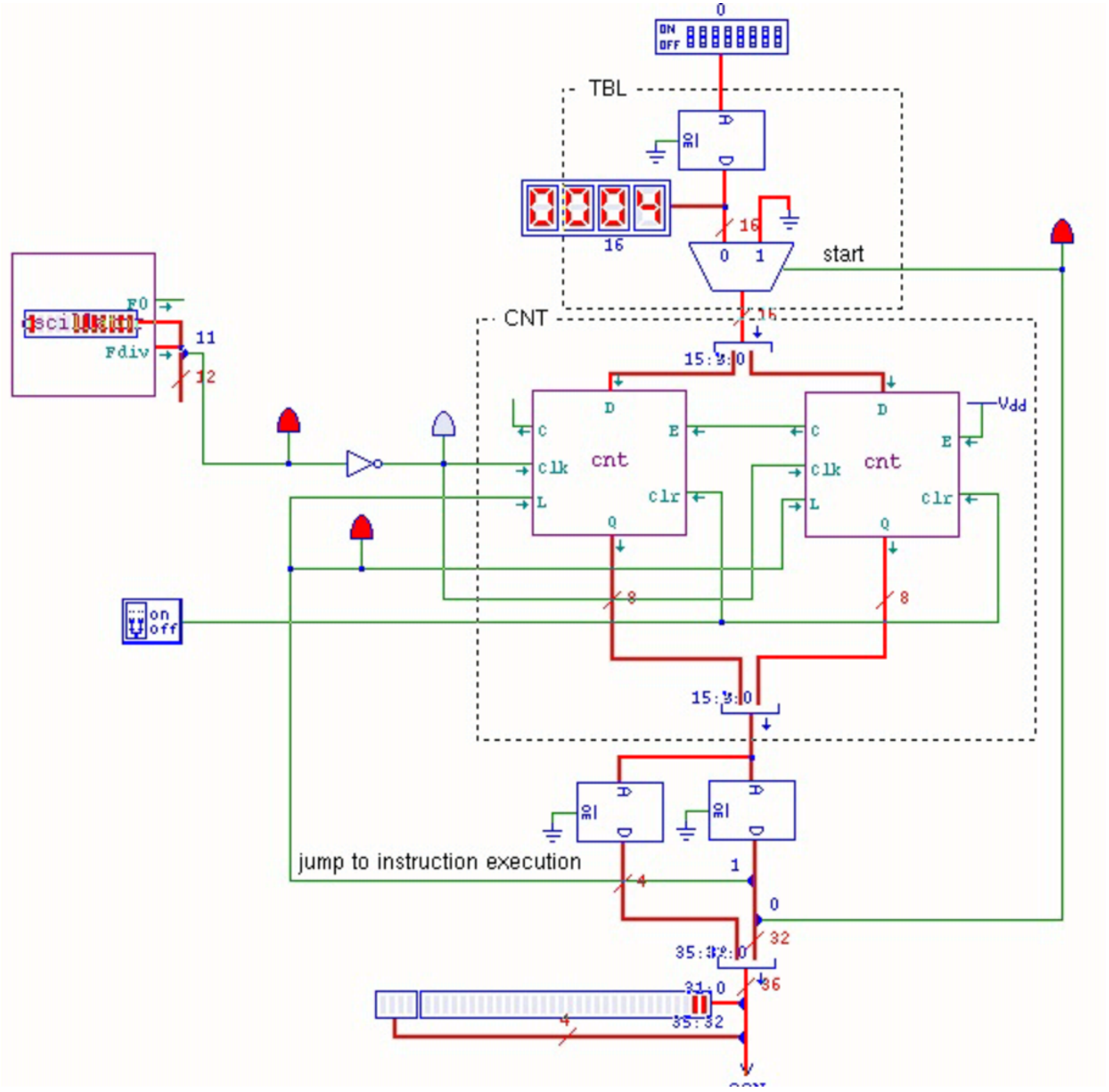


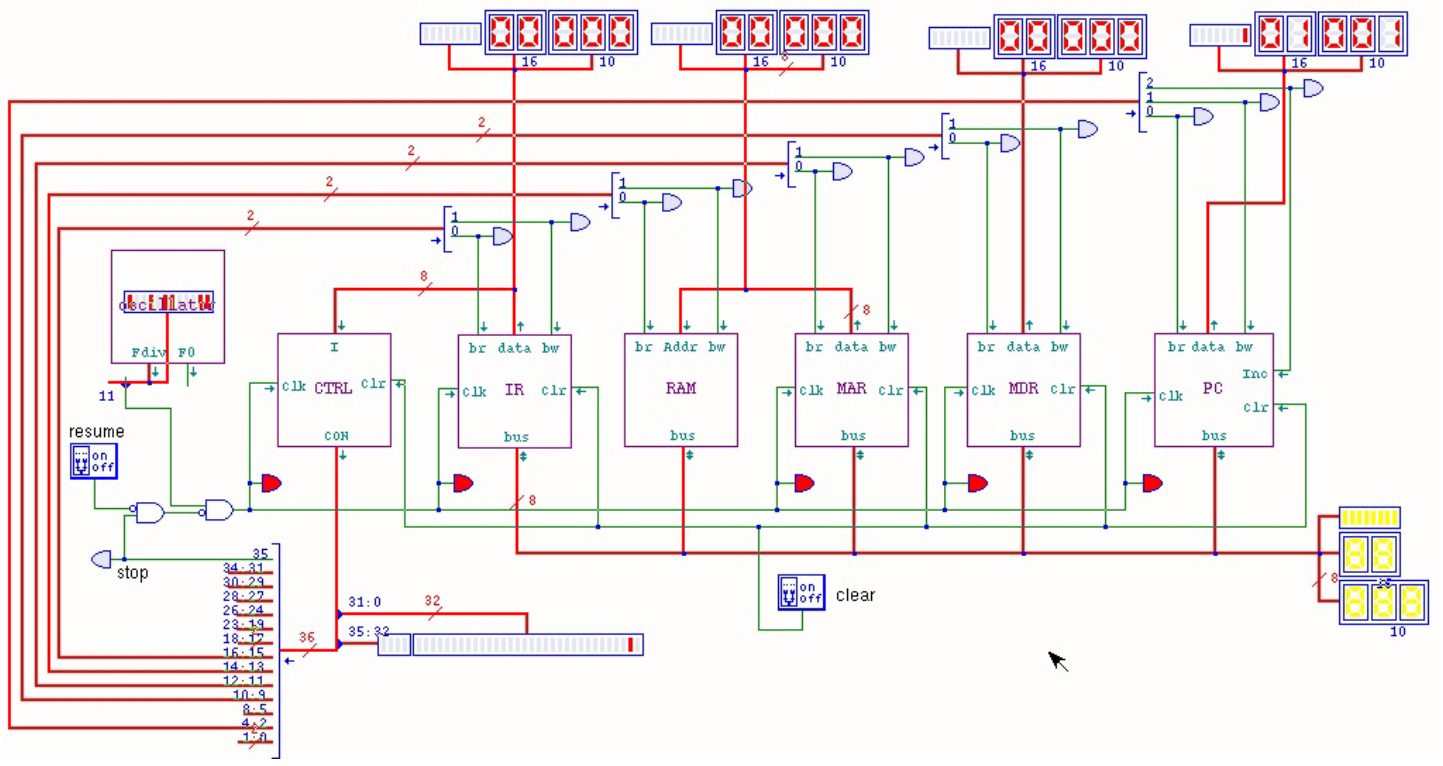
Figure 8.17. Final phase: the counters of the control unit have been increased and point to the next microinstruction. Since the original instruction (**not_operate**) did not require the performance of any operation in the data bus, the current microinstruction requires to the microcode counter group to reset to zero. The reset is done by loading the counters with zero through the multiplexer that controls the input of these counters.



After resetting the counters of the control unit, it all starts from the initial microcode

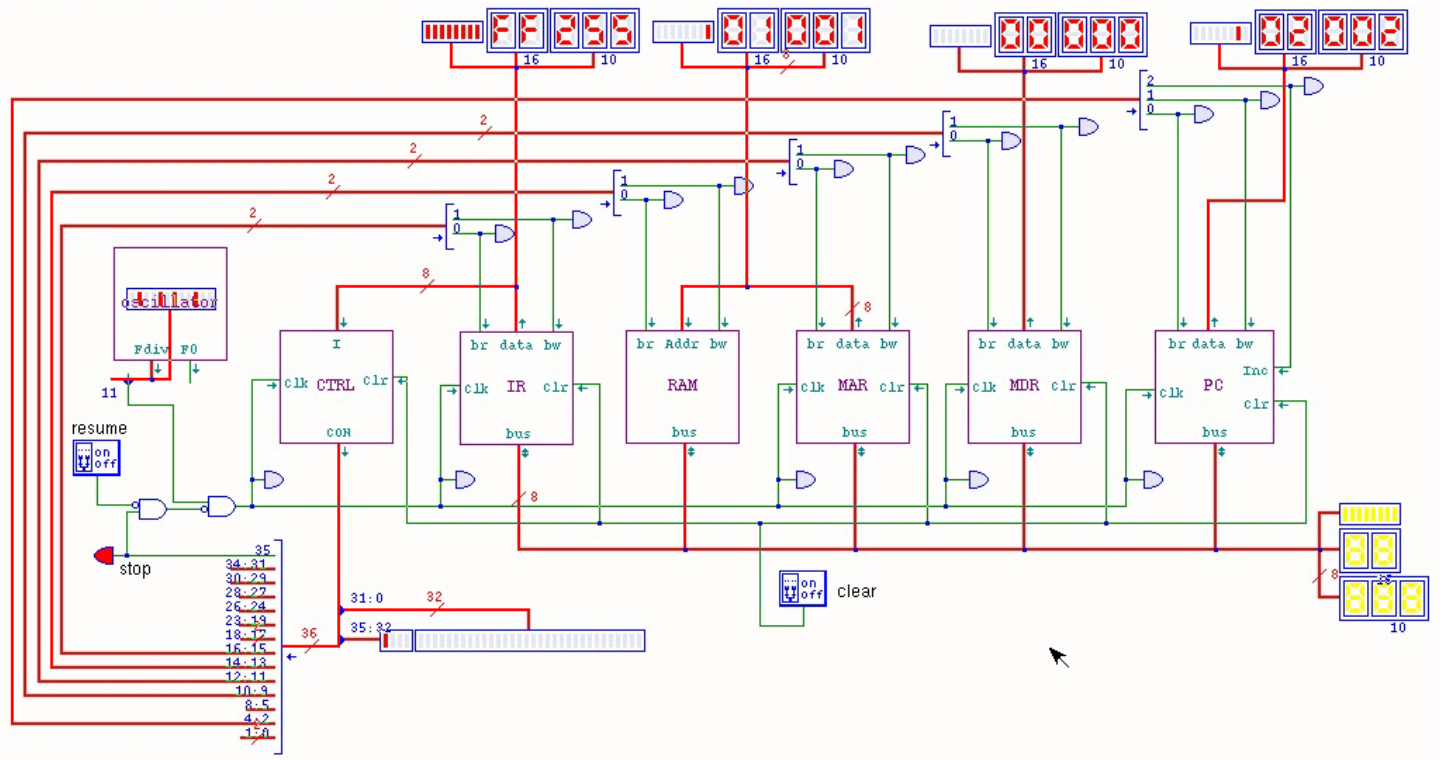
(the first three phases) which requires to do load a new instruction. During the fourth phase (jump to the microcode execution of the requested opcode) and during the final phase (jump to the microcode implementing the initial fetch cycle), inside the data bus nothing happens.

Figure 8.18. During the fourth phase (jump to the microcode execution of the requested opcode) and during the final phase (jump to the microcode implementing the initial fetch cycle), inside the data bus nothing happens.



To stop the operation of the described circuit, there is the **`stop`** instruction (11111111_2), which stops the clock signal. The figure below shows this situation.

Figure 8.19. The data bus when the instruction **`stop`** is executed: the control line CON_{35} is activated and blocks clock signal. To resume execution from that point, overcoming the stop, should turn on and off the switch located near the LED that is active.



There should be available two videos that demonstrate the execution of only two instructions (macroinstructions):

1. **`not_operate`**
2. **`stop`**

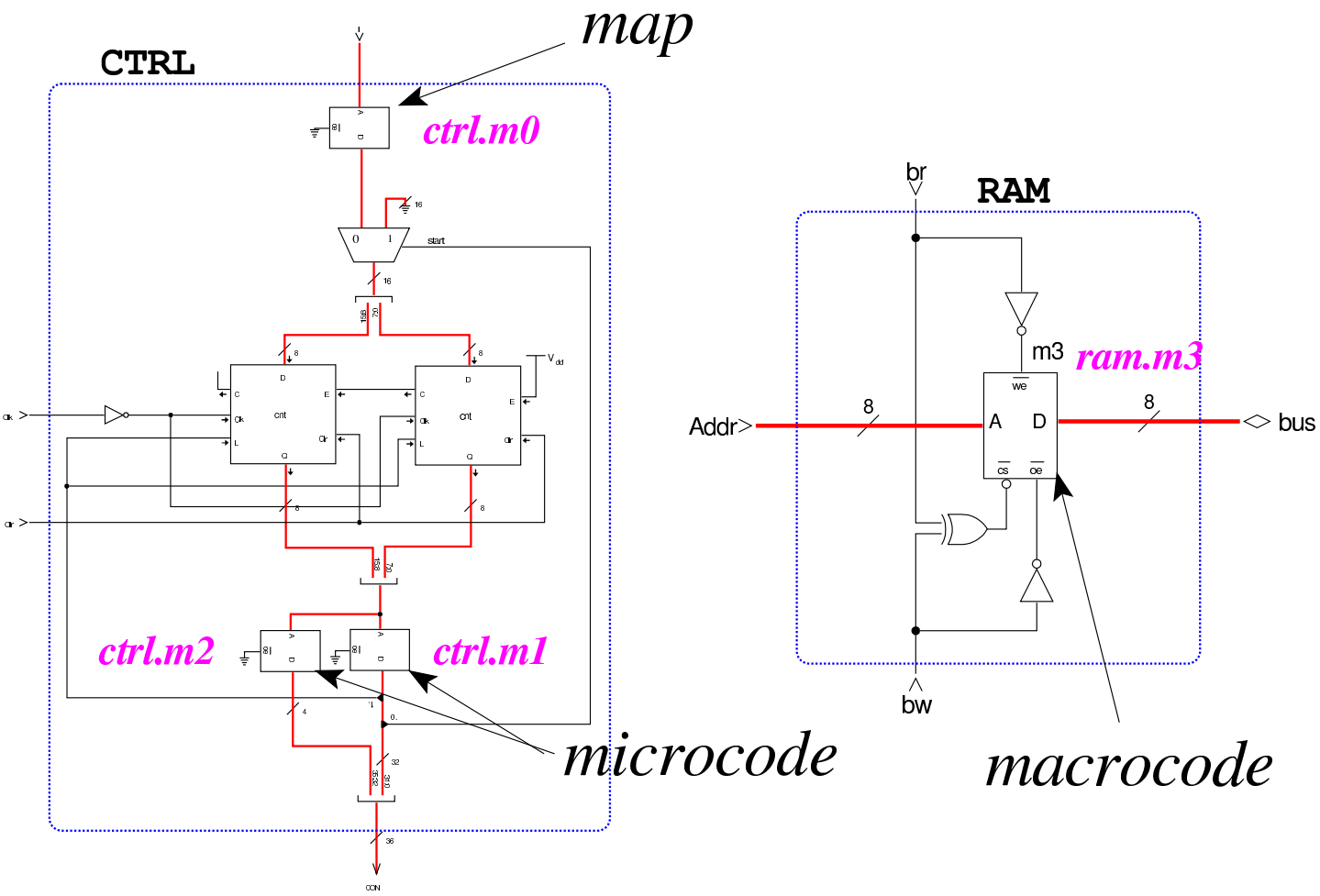
The first video ogv *http://www.youtube.com/watch?v=8MahVvyBUOo* shows what happens in the data bus, whereas the second one shows what happens inside the control unit ogv *http://www.youtube.com/watch?v=pPxCQz7IFbM* .

To describe the content of the memory banks, including the RAM, a source file is used. The source file is written according to the syntax suitable for TKGate 2 (**`gmac`** precisely). The first directives describe the memory banks, which are organized so: **`ctrl.m0`** is the first memory bank at the top of the control unit, **`ctrl.m1`** and **`ctrl.m2`** are the two memories that contain the microcode and are located at the bottom of the diagram of the control unit, **`ram.m3`** is instead contained in the **`RAM`** module of the data bus and it contains the macrocode that initially is limited only to

not_operate and **stop**.

```
map         bank[7:0]    ctrl.m0;
microcode   bank[31:0]   ctrl.m1;
microcode   bank[35:32]  ctrl.m2;
macrocode   bank[7:0]    ram.m3;
```

Figure 8.21. Where are located the memory banks.



The source file then continues with the fields that divide each microcode word (the memory cell used to represent a single microinstruction, made of **ctrl.m1** and **ctrl.m2** joined together). For example, the least significant bit is called ***ctrl_start***, while the most significant is called ***stop***. It should be noted that not all the 36 bit of the microcode word are described, because at the moment the code is limited to represent the reduction of the CPU in its first version.

```
field ctrl_start[0];     // start from address 0
field ctrl_load[1];      // load inside the counter
field pc_br[2];          // PC <-- bus
```

```
field pc_bw[3];          // PC --> bus
field pc_Inc[4];         // PC++
field mdr_br[9];         // MDR <-- bus
field mdr_bw[10];        // MDR --> bus
field mar_br[11];        // MAR <-- bus
field mar_bw[12];        // MAR --> bus
field ram_br[13];        // RAM[mar] <-- bus
field ram_bw[14];        // RAM[mar] --> bus
field ir_br[15];         // IR <-- bus
field ir_bw[16];         // IR --> bus
field stop[35];          // stop clock
```

The source file then describes the types of operands that the macroinstructions may have. It is expected to manage instruction operands without operands, or with only a single 8-bit operand. The meaning of the syntax used to describe the type **op_0** and the type **op_1** should be intuitive: the former means no argument, while the latter means the presence of a single numeric 8-bit argument.

```
operands op_0 {
    //
    // [........]
    //
    -  = { };
};
operands op_1 {
    //
    // [........][nnnnnnnn]
    //
    #1 = { +1=#1[7:0]; };
};
```

There are then the opcode descriptions. For example, the instruction **not_operate** is equal to the zero opcode (00000000_2), while the jump instruction has the opcode 15 (00001111_2). It should be noted that in the first case (**not_operate**) there are no arguments, while in the second one an argument is required.

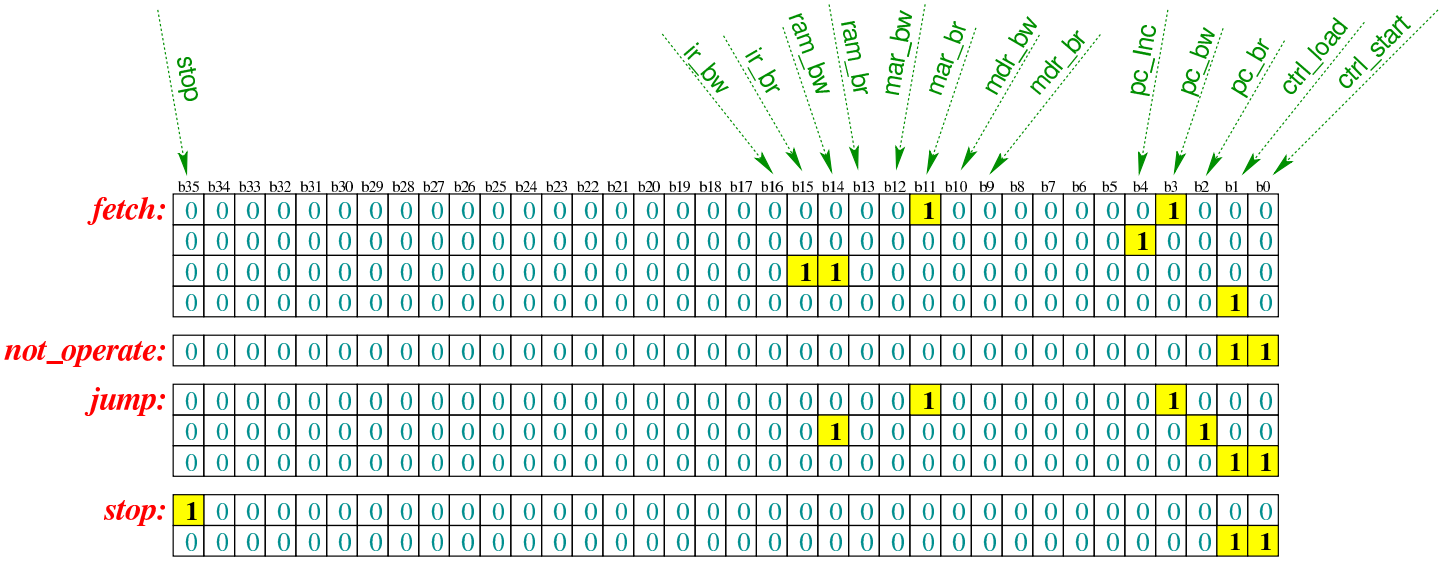
```
op not_operate {
  map not_operate : 0;
  +0[7:0]=0;
  operands op_0;
};
op jump {
  map jump : 15;                          // jump to #nn
  +0[7:0]=15;
  operands op_1;
};
op stop {
  map stop : 255;                         // stop
  +0[7:0]=255;
  operands op_0;
};
```

Then begins the microcode definition, which is placed starting from the first microcode word (address zero of the pair of memories **ctrl.m1** and **ctrl.m2**). It begins from the description of the fetch and it ends with a jump to the microcode associated to the loaded opcode. Then, at the end of the description of each opcode (in the form of microcode), it is required to jump back to the first microinstruction, which repeats the fetching cycle.

```
begin microcode @ 0
//
fetch:
    mar_br pc_bw;                     // MAR = PC
    pc_Inc;                           // PC++
    ir_br ram_bw;                     // IR = RAM[MAR]
    ctrl_load;                        // jump to the
                                      // microcode implementing
                                      // the loaded opcode
not_operate:
    ctrl_start ctrl_load;             // jump to a new fetch
//
```

```
jump:
    mar_br pc_bw;                    // MAR = PC
    pc_br ram_bw;                    // PC <-- RAM[MAR]
    ctrl_start ctrl_load;            // jump to a new fetch
//
stop:
    stop;                            // stop clock
    // If the clock signal is
    // resumed manually:
    ctrl_start ctrl_load;            // jump to a new fetch
//
end
```

Figure 8.26. Match between the contents of the memory that represents the microcode (the couple of memory banks **m1** and **m2** inside the control unit) and the microcode description itself.



Finally, it starts the macrocode, or assembler code, which should be placed inside the RAM memory:

```
begin macrocode @ 0
start:
        not_operate
        stop
end
```

Figure 8.28. Macrocode contained inside the RAM memory. The memory cells marked with "xx" have an unspecified and unknown value.

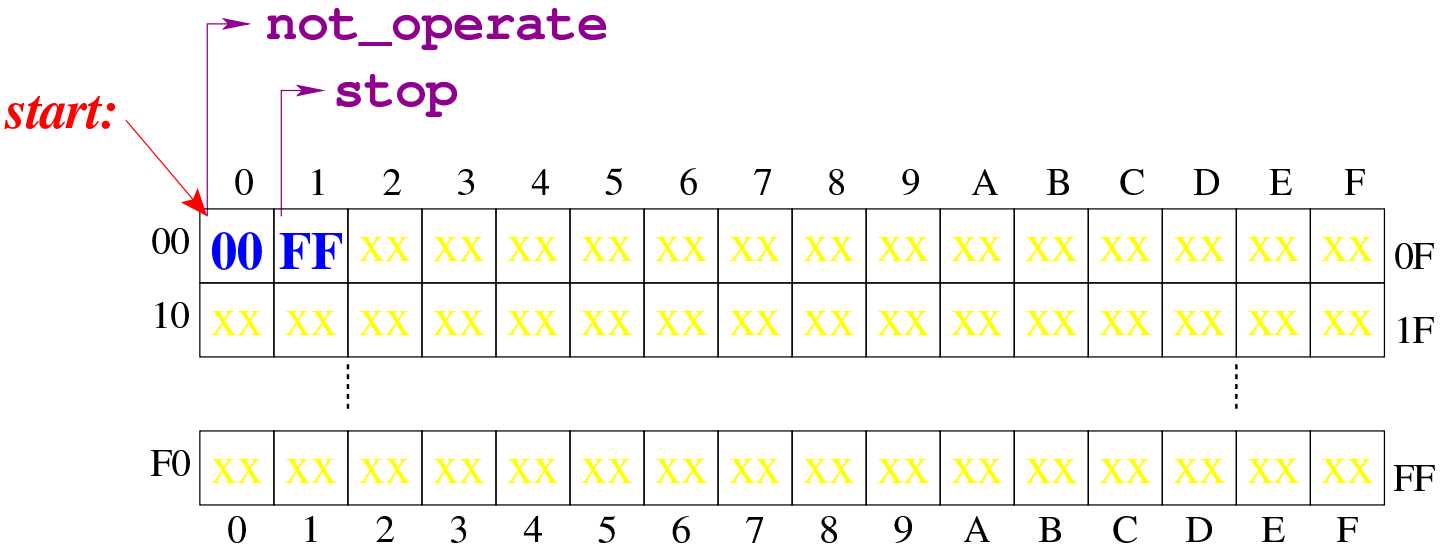


Table 8.29. Macroinstructions list for this first version of the CPU.

Syntax	Description
`not_operate`	Do nothing, just go to the next instruction.
`jump #`***ref***	Jump to the instruction that is located at the specified address in memory.
`stop`	Stop the clock.

The source file described above should be available at *attachments/xcpu/xcpu-a1.gm* . If follows the procedure to compile it with **gmac20**, the TKGate 2 code compiler:

```
$ gmac20 -o x.mem -m x.map ↩
↪        xcpu-a1.gm [Enter]
```

The compilation produces the file '`x.mem`' that contains the images for all the memory banks. It is a text file and it looks like this:

```
@memory ctrl.m1

@0
808 10 c000 2 3 808 4004 3
0 3
```

```
@memory ctrl.m2

@0
0 0 0 0 0 0 0 0
8 0

@memory ctrl.m0

@0
4

@f
5

@ff
8

@memory ram.m3

@0
0 ff
```

The compilation produces also the file '`x.map`' that contains the microcode and macrocode maps; that is, the addresses of known labels:

```
Microcode Symbols (total=4):
  ------------------------------------
  fetch                          00000000
  jump                           00000005
  not_operate                    00000004
  stop                           00000008

Macrocode Symbols (total=1):
  Global         Local
```

```
   ------------------------------------
   start                              00000000
```

The file '`x.mem`', produced by the compilation, is used by TKGate 2 to load the memory banks initial contents. But an additional script is necessary:

```
module LM();
  initial
    begin
      $readmemh ("x.mem");
    end
endmodule
```

This is a simple module —called **`LM`** inside the TKGate Verilog netlist source— that should be placed somewhere, just to make it work at the beginning of the TKGate simulation. The script is responsible to load the file '`x.mem`' to the memory banks.

The TKGate 2 Verilog netlist source should be available at *attachments/xcpu/xcpu-a1.v* and TKGate might be used to experiment with this CPU version with the following command:

`$` **`tkgate20 xcpu-a1.v`** [*Enter*]

This first CPU version has also another instruction, not yes experimented: **`jump`**. The jump is done with the following steps:

1. ***MAR*** = ***PC***
2. ***PC*** = ***RAM***[***MAR***]

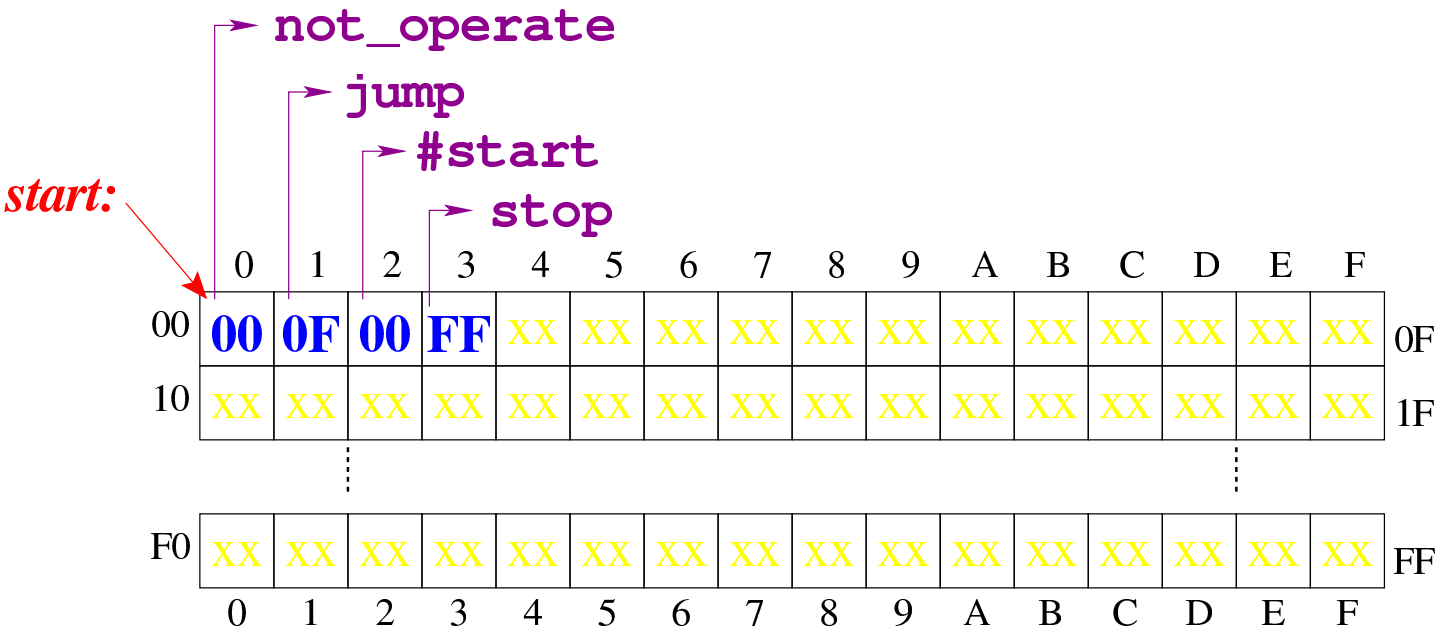
That is: the ***MAR*** register receives a copy of the ***PC*** register, which contains the address following the **`jump`** opcode. At the ***MAR*** address, the memory contains (should contain) another address, which is copied back to the ***PC*** register (and no increment is applied to the ***PC*** register).

The **`jump`** opcode precedes an argument: a number representing a memory address, which is the place where the ***PC*** register should point, so that the next instruction would be that one located at the new address position. When the argument is read, it is not sent to the ***IR*** register, because it is not an opcode.

The following macrocode example shows an infinite loop. Please notice that the TKGate compiler transforms the string **#start** into the address matching the label **start** inside the RAM memory: as the label **start** appears at the first position, it matches to the address zero.

```
begin macrocode @ 0
start:
        not_operate
        jump #start
        stop
end
```

Figure 8.34. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.



As it is an infinite loop, the **stop** is never reached. The following figures show what happens on the data bus after the instruction **not_operate** is executed.

Figure 8.35. The **`jump`** opcode is loaded and the ***PC*** register is pointing to the address following it; the ***MAR*** register is loading the memory content at the address pointed by ***PC***.

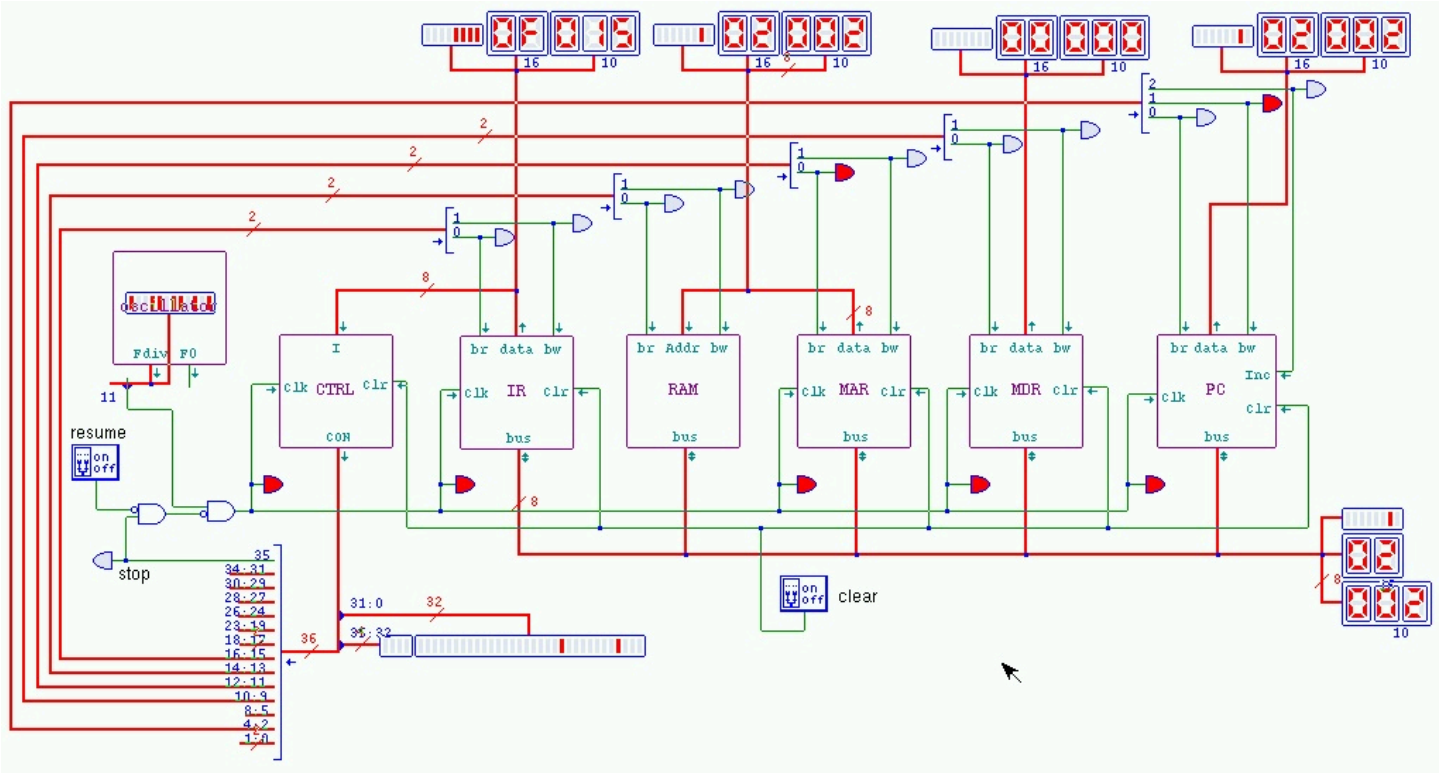
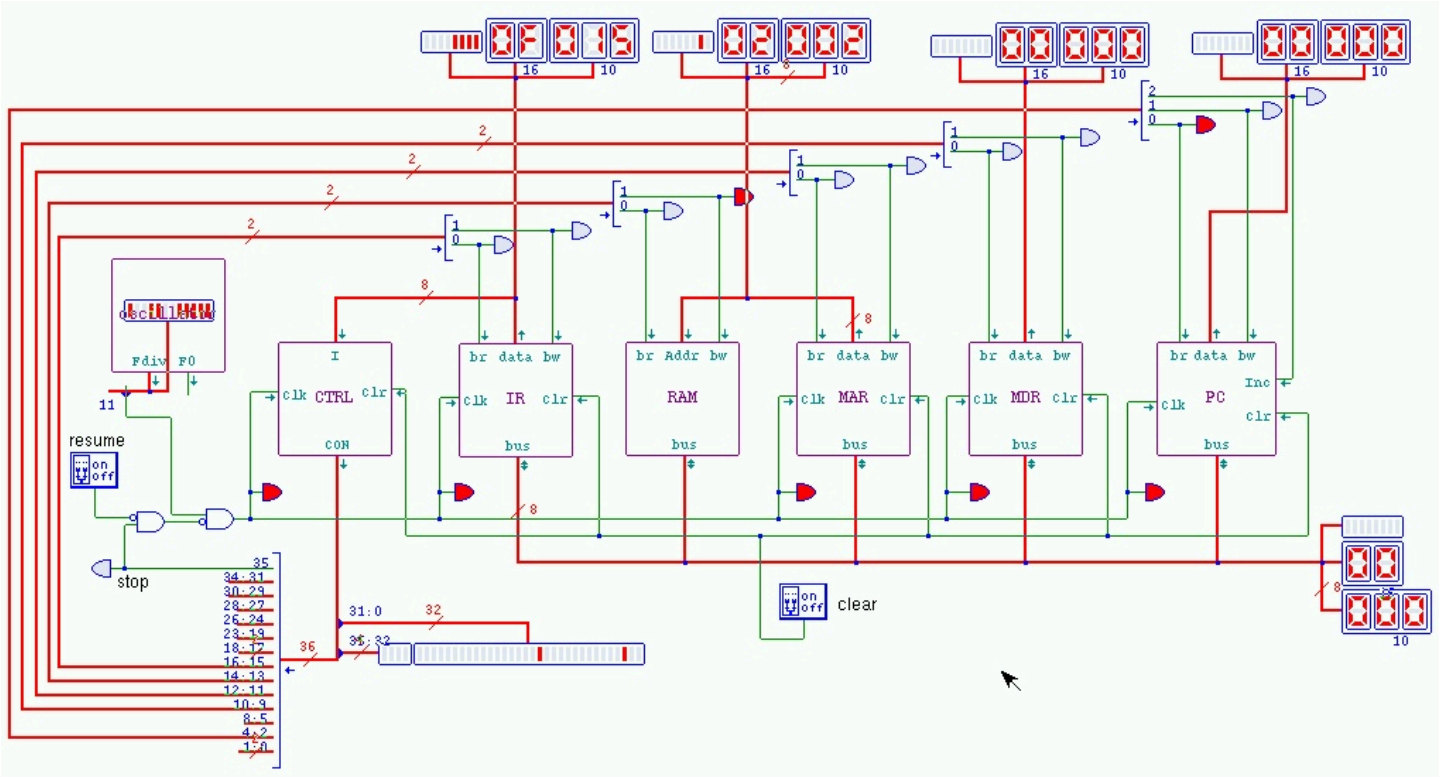


Figure 8.36. The ***PC*** register is loaded with the value contained in memory at the address pointed by the ***MAR*** register.

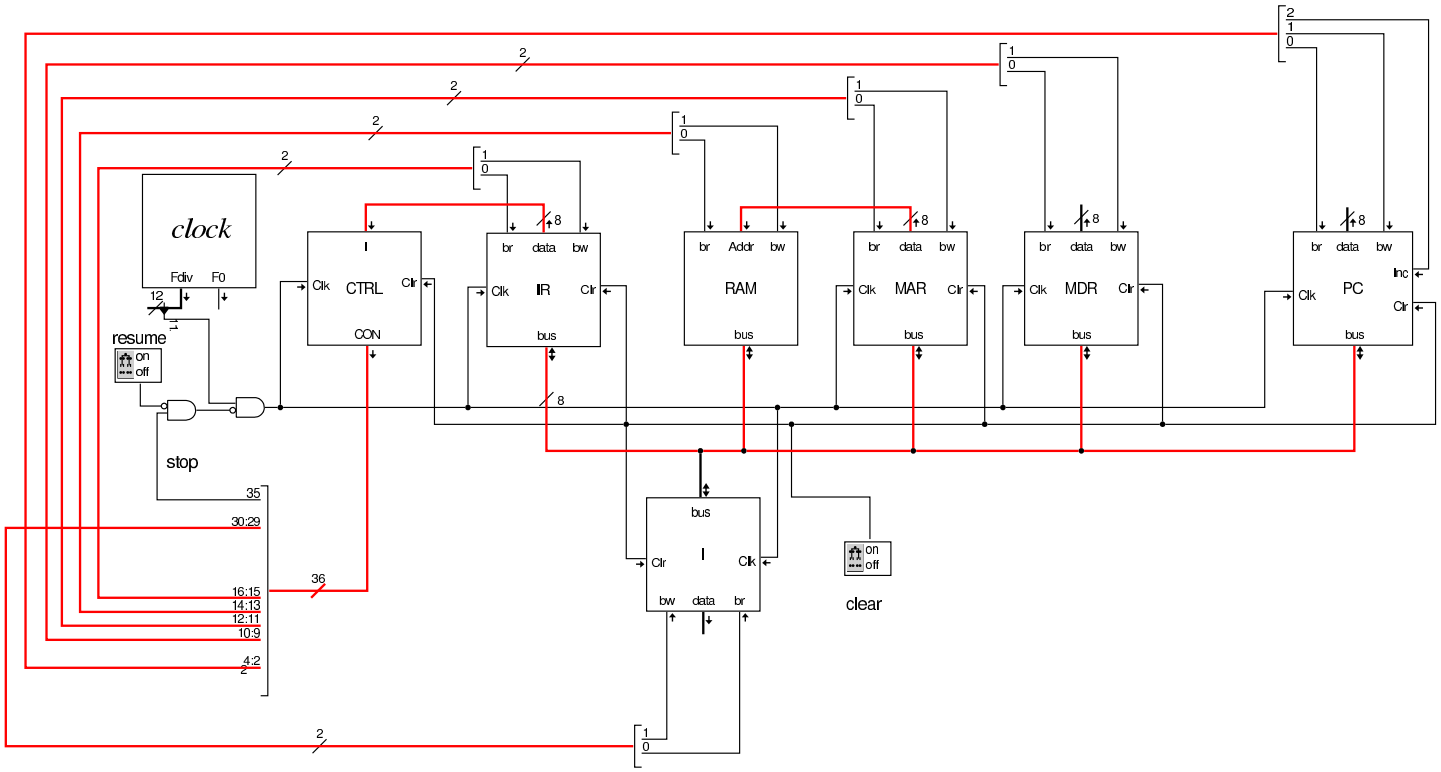


There should be available a video showing the example above: ogv *http://www.youtube.com/watch?v=Z8bTO8WjYYc* .

8.2 Version "A2": memory index

The second CPU subversion adds only a special register, named ***I*** (*index*), which should contain a memory index. It should be used to read and write the RAM memory through an index that could be managed. The ***I*** register is made the same way as ***MDR***, ***MAR*** and ***IR***.

Figure 8.37. Simple CPU version "A2".



Inside the code that describes the microcode word fields, the following fields are added: they are used specifically to manage the ***I*** register:

```
field i_br[29];          // I <-- bus
field i_bw[30];          // I --> bus
```

Inside the opcode list there are new instructions and the same is for the microcode description:

```
op load_ref
{
  map load_ref : 1;                    // load from address #nn
  +0[7:0]=1;
  operands op_1;
};
op load_reg
{
  map load_reg : 2;                    // load from address %I
  +0[7:0]=2;
  operands op_0;
```

```
};
op store_ref {
  map store_ref : 3;                    // store to address #nn
  +0[7:0]=3;
  operands op_1;
};
op store_reg {
  map store_reg : 4;                    // store to address I
  +0[7:0]=4;
  operands op_0;
};
op move_mdr_i {
  map move_mdr_i : 11;                  // move MDR to I
  +0[7:0]=11;
  operands op_0;
};
op move_i_mdr {
  map move_i_mdr : 12;                  // move I to MDR
  +0[7:0]=12;
  operands op_0;
};
```

```
begin microcode @ 0
…
load_ref:
    mar_br pc_bw;                   // MAR <-- PC
    pc_Inc;                         // PC++
    // The RAM memory does not have a clock,
    // thus, it is not possible to do just
    // MAR <-- RAM[MAR]
    i_br ram_bw;                    // I <-- RAM[MAR]
    mar_br i_bw;                    // MAR <-- I
    mdr_br ram_bw;                  // MDR <-- RAM[MAR]
    ctrl_start ctrl_load;           // CNT <-- 0
```

```
//
load_reg:
    mar_br i_bw;                    // MAR <-- I
    mdr_br ram_bw;                  // MDR <-- RAM[MAR]
    ctrl_start ctrl_load;           // CNT <-- 0
//
store_ref:
    mar_br pc_bw;                   // MAR <-- PC
    pc_Inc;                         // PC++
    i_br ram_bw;                    // I <-- RAM[MAR]
    mar_br i_bw;                    // MAR <-- I
    ram_br mdr_bw;                  // RAM[MAR] <-- MDR
    ctrl_start ctrl_load;           // CNT <-- 0
//
store_reg:
    mar_br i_bw;                    // MAR <-- I
    ram_br mdr_bw;                  // RAM[MAR] <-- MDR
    ctrl_start ctrl_load;           // CNT <-- 0
//
move_mdr_i:
    i_br mdr_bw;                    // I <-- MDR
    ctrl_start ctrl_load;           // CNT <-- 0
//
move_i_mdr:
    mdr_br i_bw;                    // MDR <-- I
    ctrl_start ctrl_load;           // CNT <-- 0
…
end
```

Figure 8.41. Match between the contents of the memory that represents the microcode (the couple of memory banks **m1** and **m2** inside the control unit) and the microcode description itself.

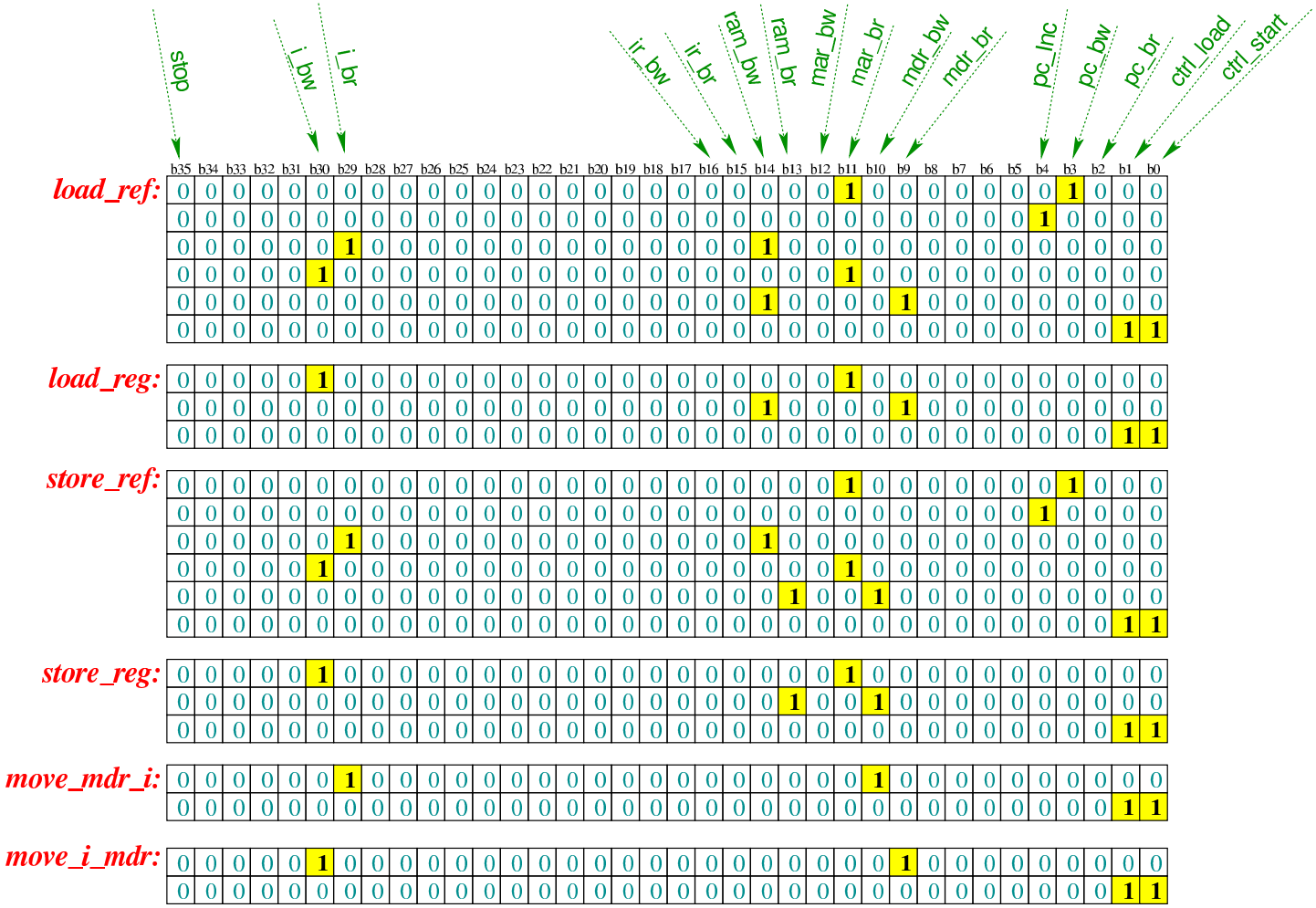


Table 8.42. Macroinstructions added for this version of the simple CPU.

Syntax	Description
`load_ref #`***ref***	Load reference: it loads the value contained in memory, at the specified address, into the ***MDR*** register. The ***I*** register loads the argument.
`load_reg`	Load register: it loads inside the ***MDR*** register the value contained in memory, at the address pointed by the ***I*** register.
`store_ref #`***ref***	Store reference: it copies into the memory, at the address specified by the argument, the value contained inside the ***MDR*** register. The ***I*** register loads the argument.

Syntax	Description
`store_reg`	Store register: it copies into the memory, at the address pointed by the ***I*** register, the value contained inside the ***MDR*** register.
`move_mdr_i`	It copies the ***MDR*** register into the ***I*** register.
`move_i_mdr`	It copies the ***I*** register into the ***MDR*** register.

8.2.1 Loading instructions

«

As a first example, the following macrocode is proposed:

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_i
        load_reg
stop:
        stop
data_1:
       .byte 3
end
```

The address matching the label **`data_1`** is loaded into the ***MDR*** register (it is the address 5). Then, ***MDR*** is copied into ***I*** (it is the value 3). Then, the value contained in memory at the address pointed by ***I*** is copied to ***MDR***. At the address 3 there is the instruction **`load_reg`** which has the opcode 2: that is, at the end the ***MDR*** register contains the value 2. The full source should he available at *attachments/xcpu/xcpu-a2.gm* .

Figure 8.44. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

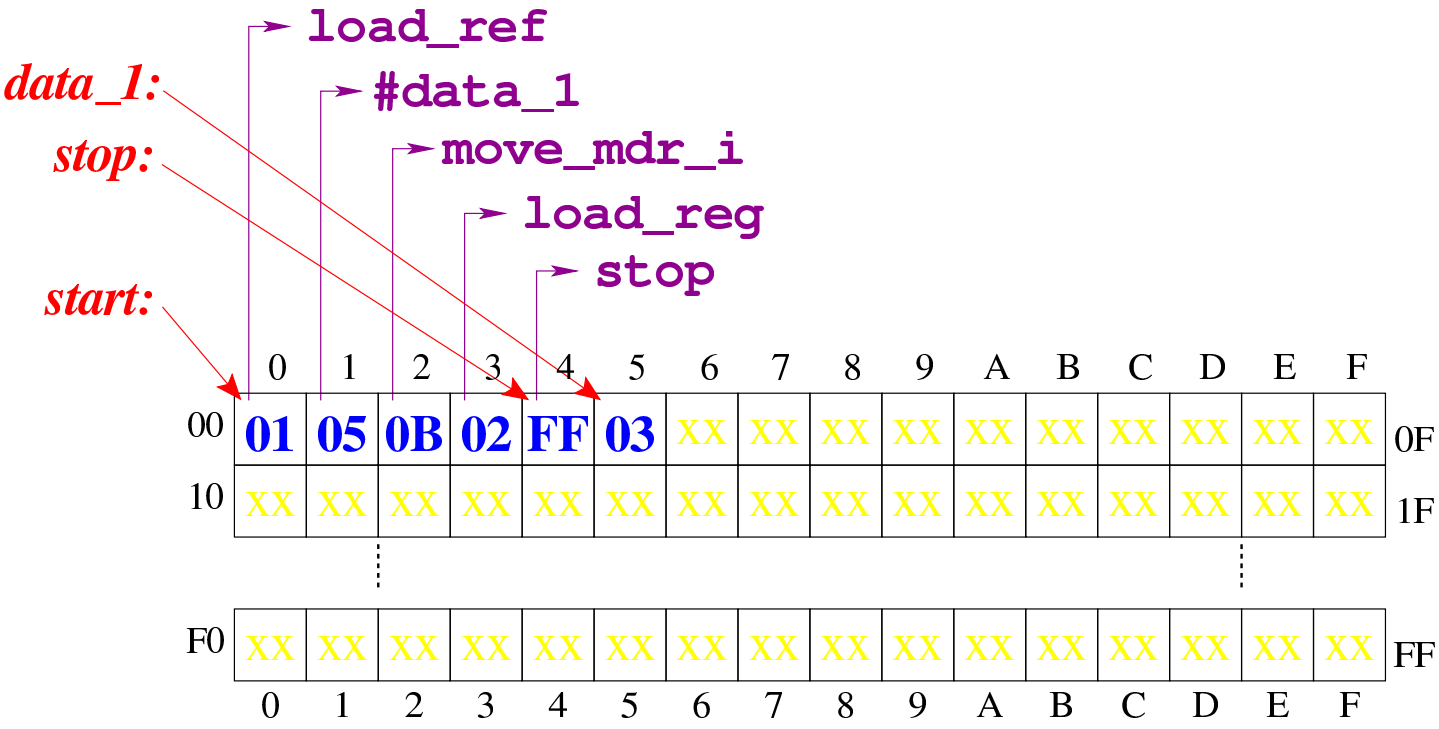
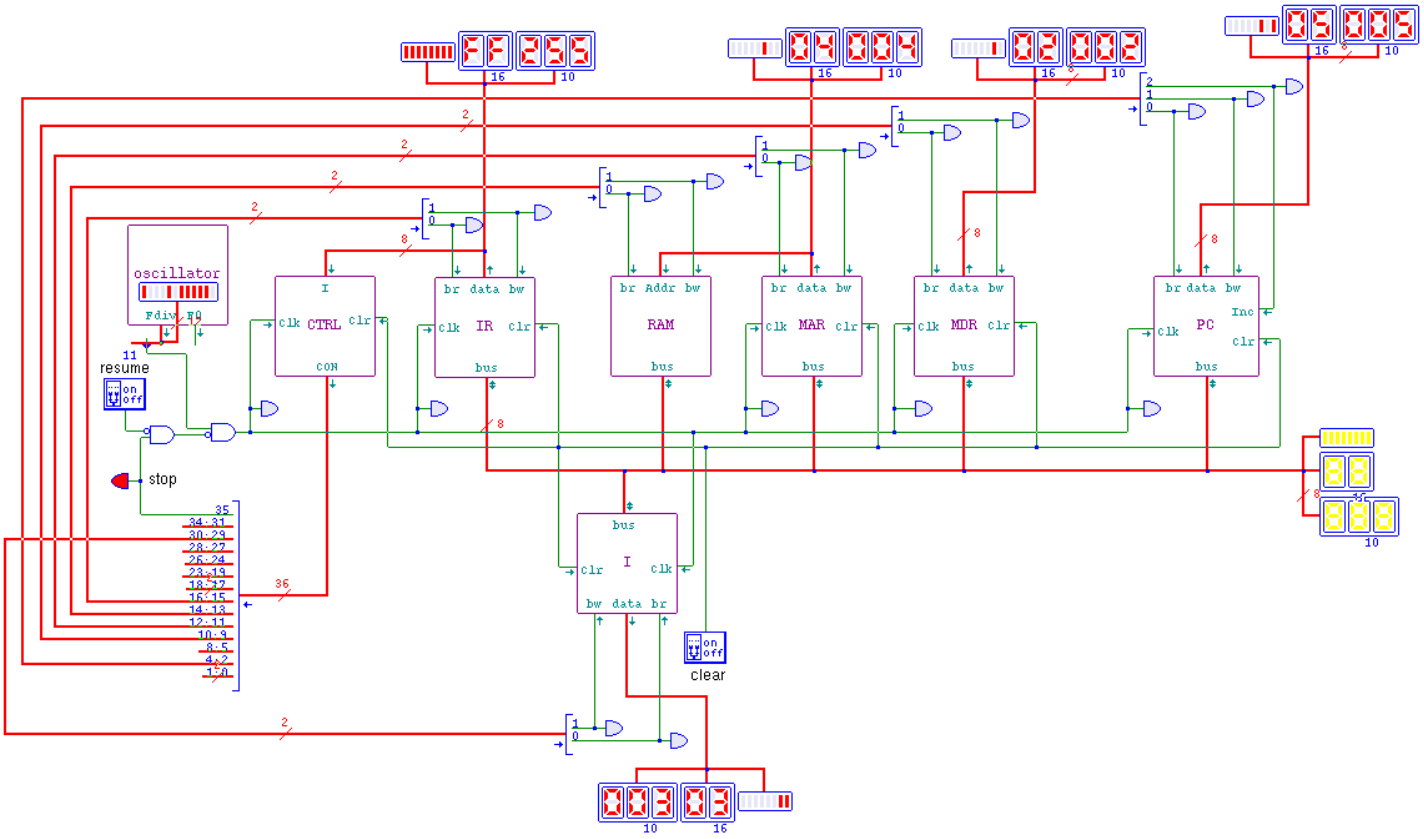


Figure 8.45. Data bus after the execution of the load example above. Video: ogv *http://www.youtube.com/watch?v=AXUSrH49cF49w*



8.2.2 Storing instructions

The following example shows the usage of the instructions **store_ref** and **store_reg**:

```
begin macrocode @ 0
start:
        load_ref #data_1
        store_ref #data_2
        move_mdr_i
        store_reg
stop:
        stop
data_1:
       .byte 15
data_2:
       .byte 0
```

```
end
```

The ***MDR*** loads the value contained in memory at the address matching the label **`data_1`** (that is, it loads the value 15). Then, the ***MDR*** content is written in memory at the address represented by the label **`data_2`** (that is, the value 15 is written at the address 8, replacing the previous zero). Then, the ***MDR*** is copied to ***I*** (that is, the ***I*** register will get the value 15). Then, at the address pointed by ***I***, the memory is written with the value contained inside ***MDR*** (that is, the value 15 is written at the address 15).

Figure 8.47. RAM memory before the execution.

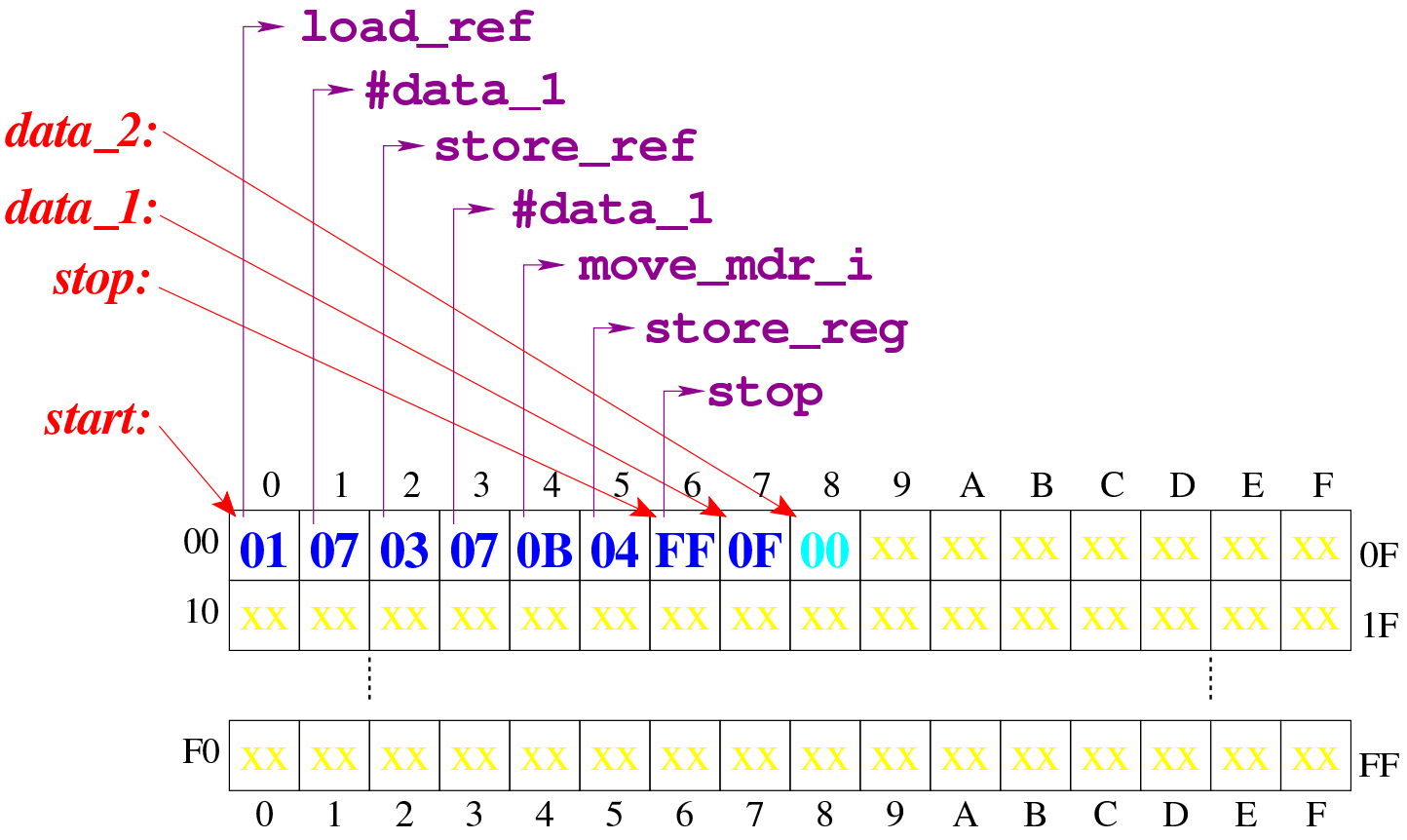


Figure 8.48. RAM memory after the execution of the **store_ref** instruction.

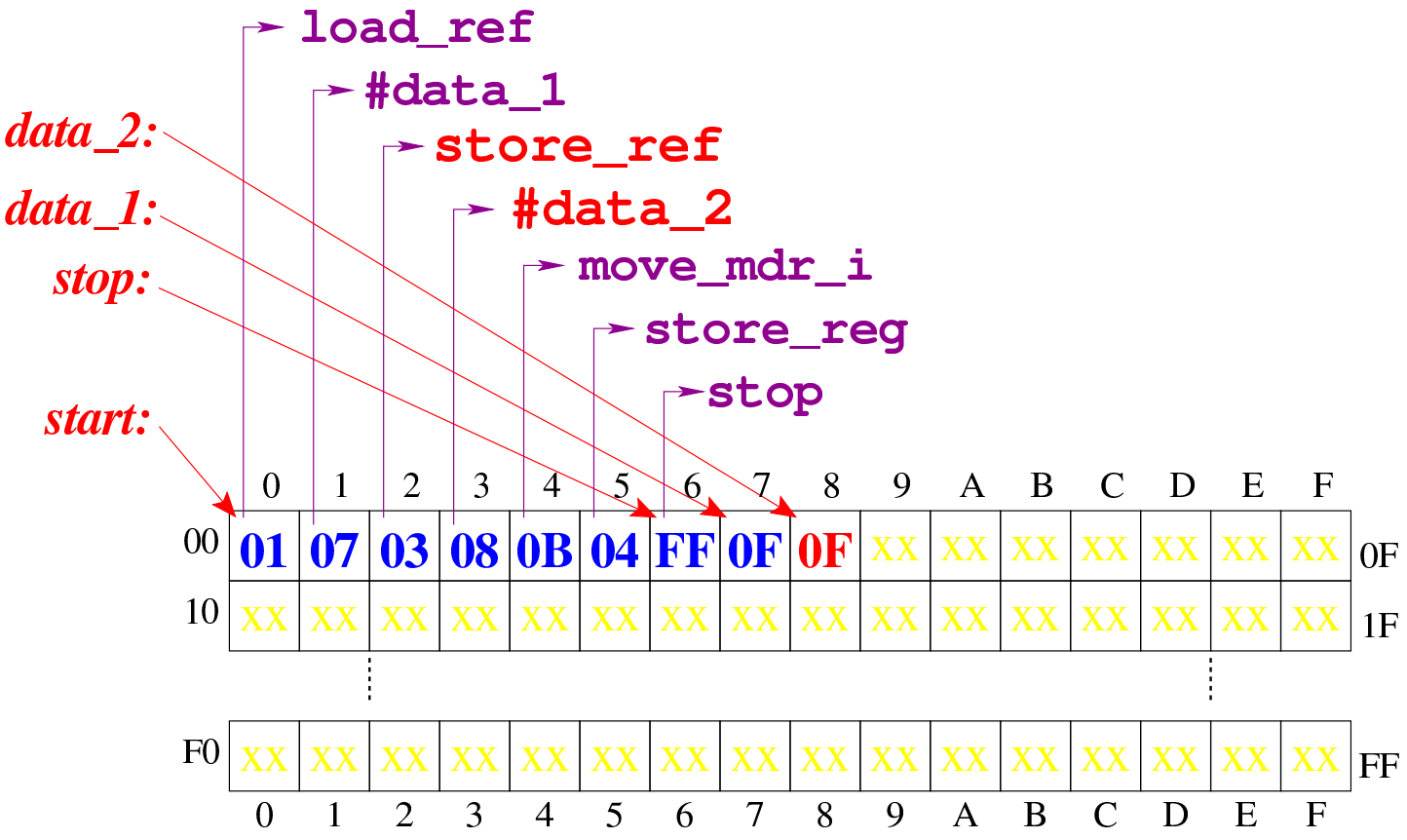


Figure 8.49. RAM memory at the end.

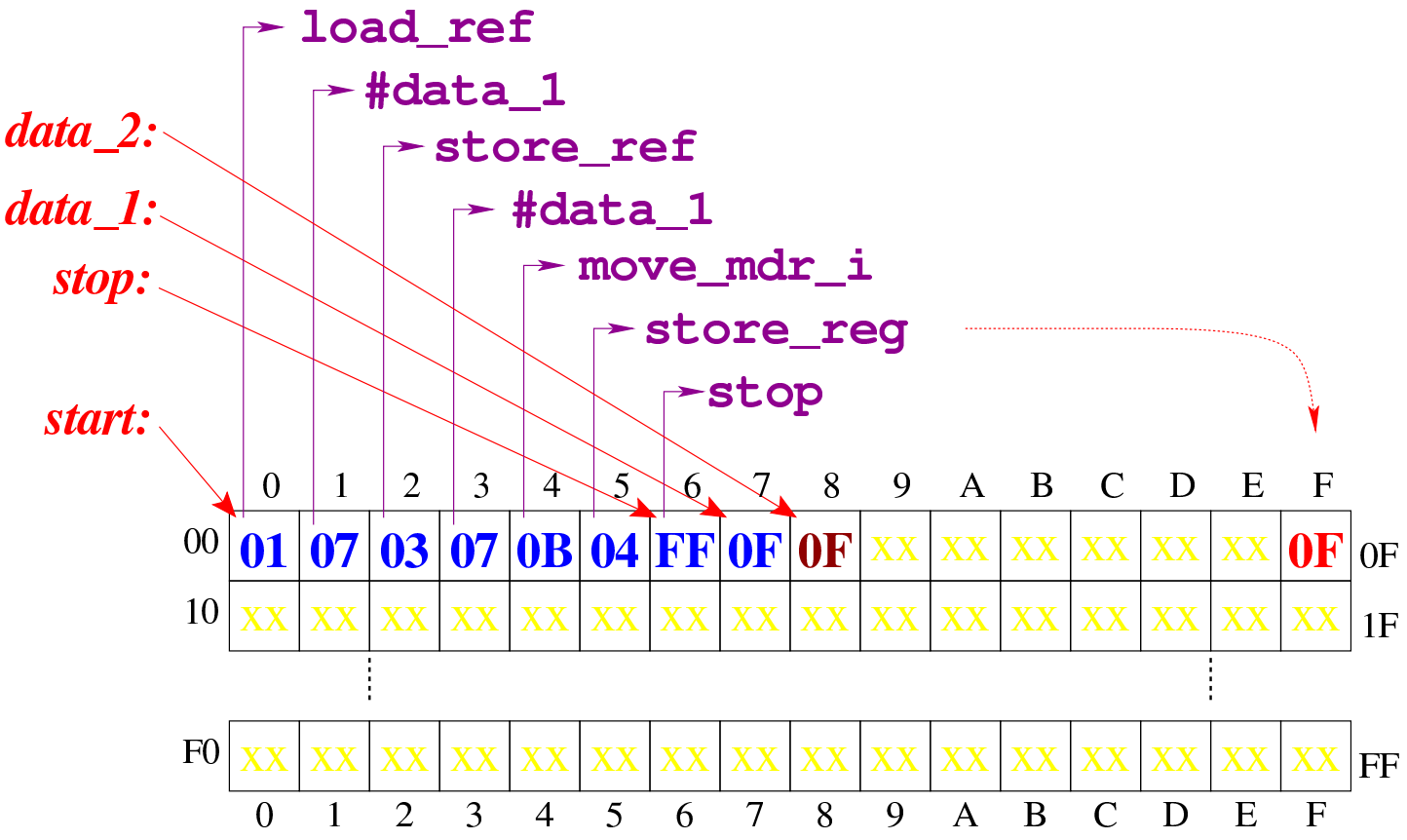
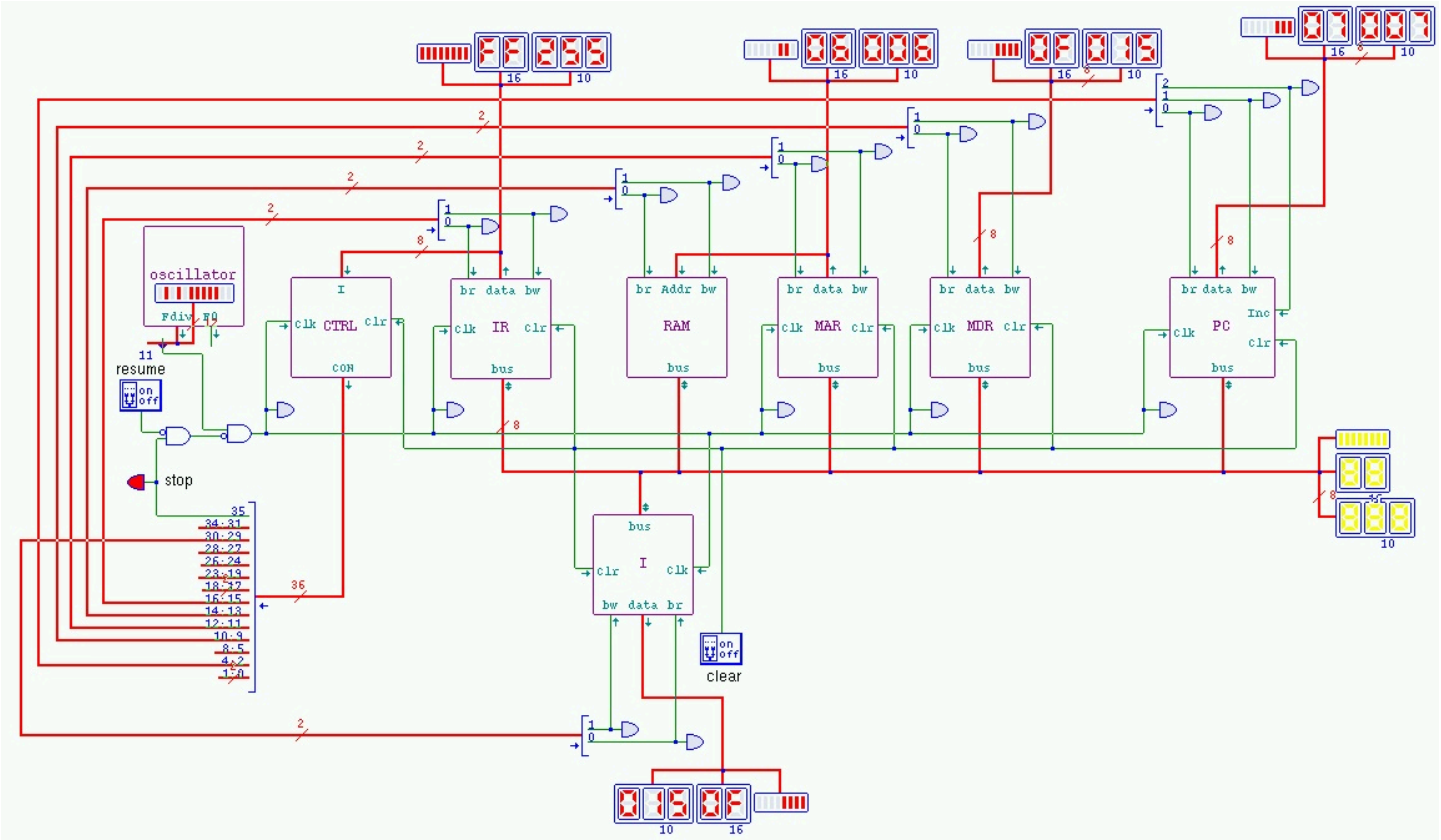


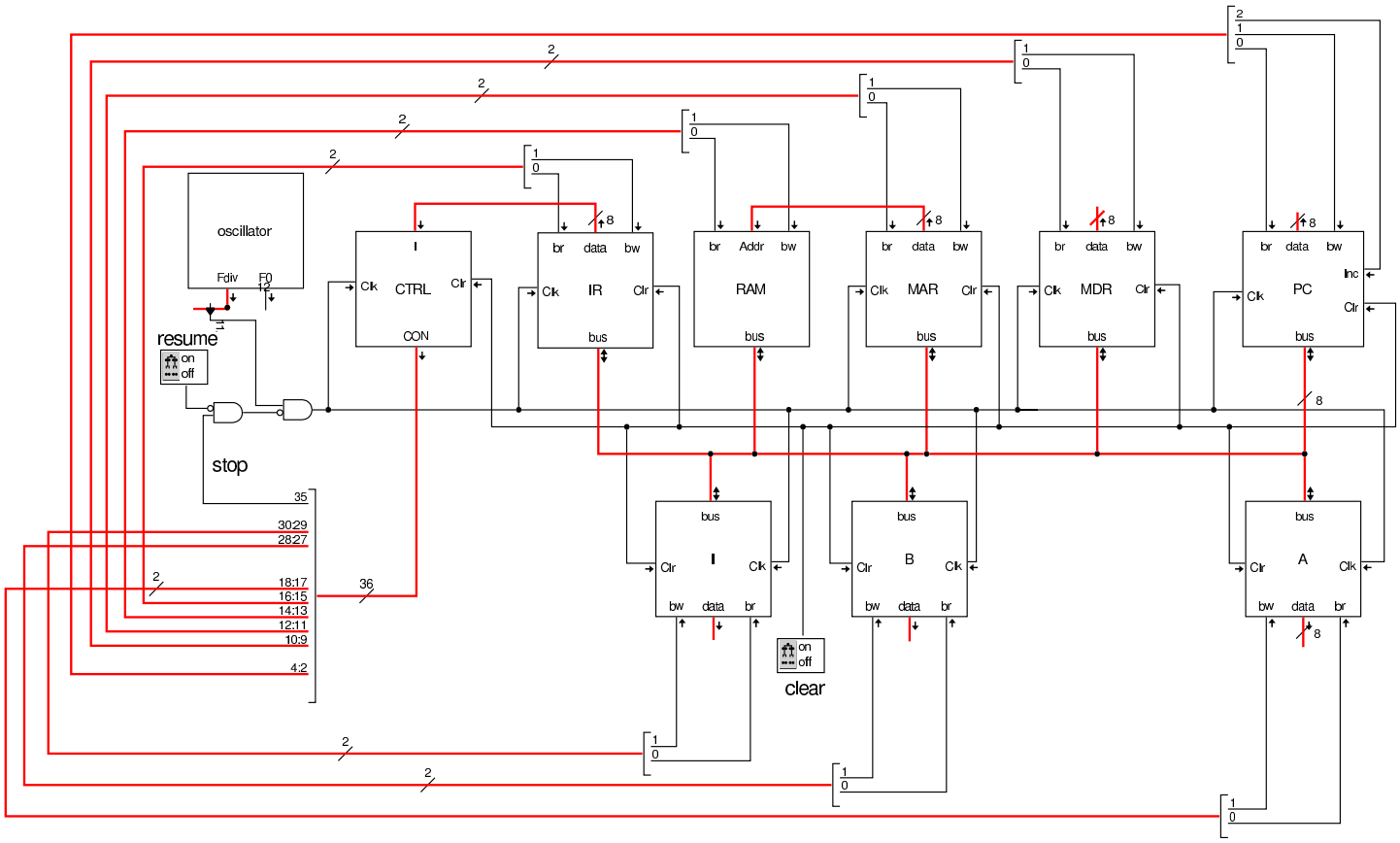
Figure 8.50. Data bus after the execution of the store example above. Video: ogv *http://www.youtube.com/watch?v=lHxx3SR56hE56*



8.3 Version "A3": general purpose registers

On the third subversion of the simple CPU two registers are added, without any particular purpose: ***A*** and ***B***. These registers are made in the same way as ***I***, ***MDR***, ***MAR*** and ***IR***.

Figure 8.51. Simple CPU version "A3".



Inside the code that describes the microcode word fields, the following fields are added: they are used specifically to manage the ***A*** and ***B*** registers:

```
field a_br[17];          // A <-- bus
field a_bw[18];          // A --> bus
field b_br[27];          // B <-- bus
field b_bw[28];          // B --> bus
```

Inside the opcode list there are new instructions and the same is for the microcode description:

```
op move_mdr_a {
  map move_mdr_a : 5;                    // move MDR to A
  +0[7:0]=5;
  operands op_0;
};
op move_a_mdr {
  map move_a_mdr : 6;                    // move A to MDR
```

```
  +0[7:0]=6;
  operands op_0;
};
op move_mdr_b {
  map move_mdr_b : 7;                    // move MDR to B
  +0[7:0]=7;
  operands op_0;
};
op move_b_mdr {
  map move_b_mdr : 8;                    // move B to MDR
  +0[7:0]=8;
  operands op_0;
};
```

```
begin microcode @ 0
…
move_mdr_a:
    a_br mdr_bw;                  // A <-- MDR
    ctrl_start ctrl_load;         // CNT <-- 0
//
move_a_mdr:
    mdr_br a_bw;                  // MDR <-- A
    ctrl_start ctrl_load;         // CNT <-- 0
//
move_mdr_b:
    b_br mdr_bw;                  // B <-- MDR
    ctrl_start ctrl_load;         // CNT <-- 0
//
move_b_mdr:
    mdr_br b_bw;                  // MDR <-- B
    ctrl_start ctrl_load;         // CNT <-- 0
…
end
```

Figure 8.55. Match between the contents of the memory that represents the microcode (the couple of memory banks **m1** and **m2** inside the control unit) and the microcode description itself.

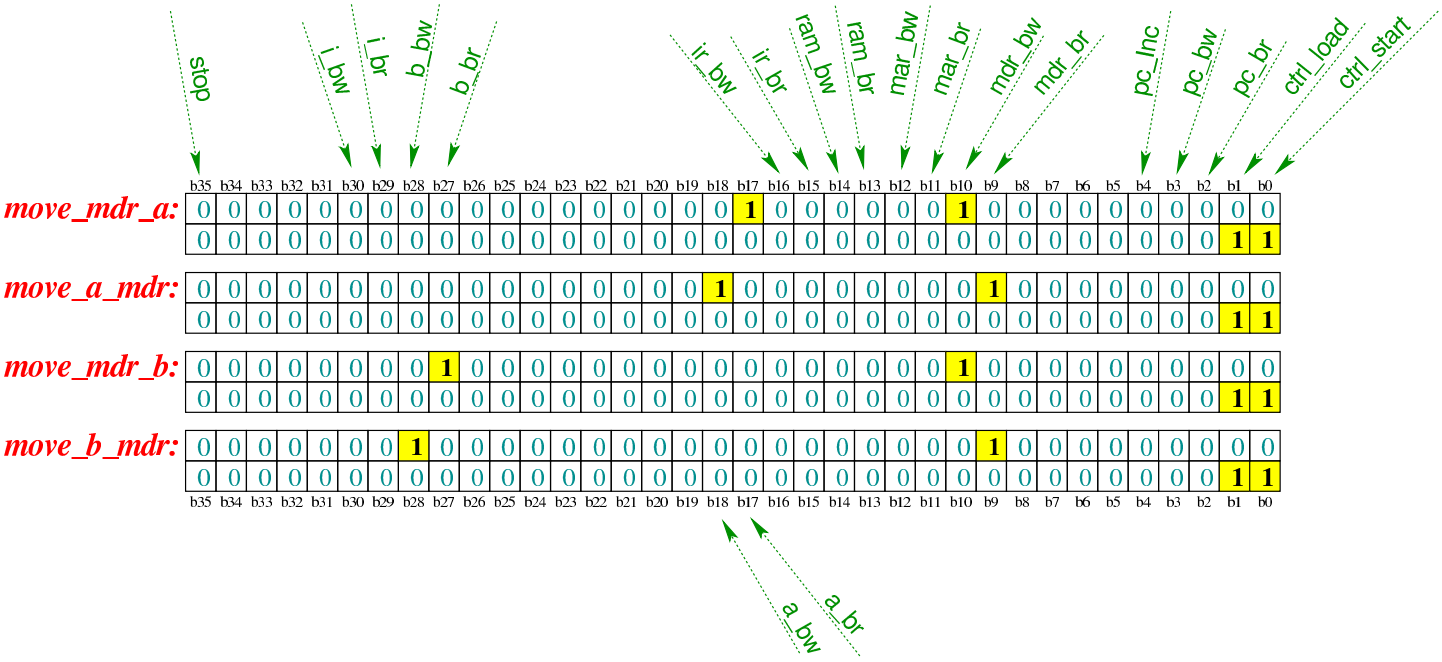


Table 8.56. Macroinstructions added for this version of the simple CPU.

Syntax	Description
`move_mdr_a`	It copies the ***MDR*** content into the ***A*** register.
`move_a_mdr`	It copies the ***A*** content into the ***MDR*** register.
`move_mdr_b`	It copies the ***MDR*** content into the ***B*** register.
`move_b_mdr`	It copies the ***B*** content into the ***MDR*** register.

It follows a macrocode example where some data is loaded from memory and stored inside the ***A*** and ***B*** registers:

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        load_ref #data_2
```

```
        move_mdr_b
stop:
        stop
data_1:
       .byte 17
data_2:
       .byte 11
end
```

It loads into the ***MDR*** register the value 17 (located in memory at the address matching the label **`data_1`**) and then it copies the loaded value into the ***A*** register. Then it does the same with the value 11 (located in memory at the address matching the label **`data_2`**) that is copied into the ***B*** register. The full source file should be available at *attachments/xcpu/xcpu-a3.gm* .

Figure 8.58. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

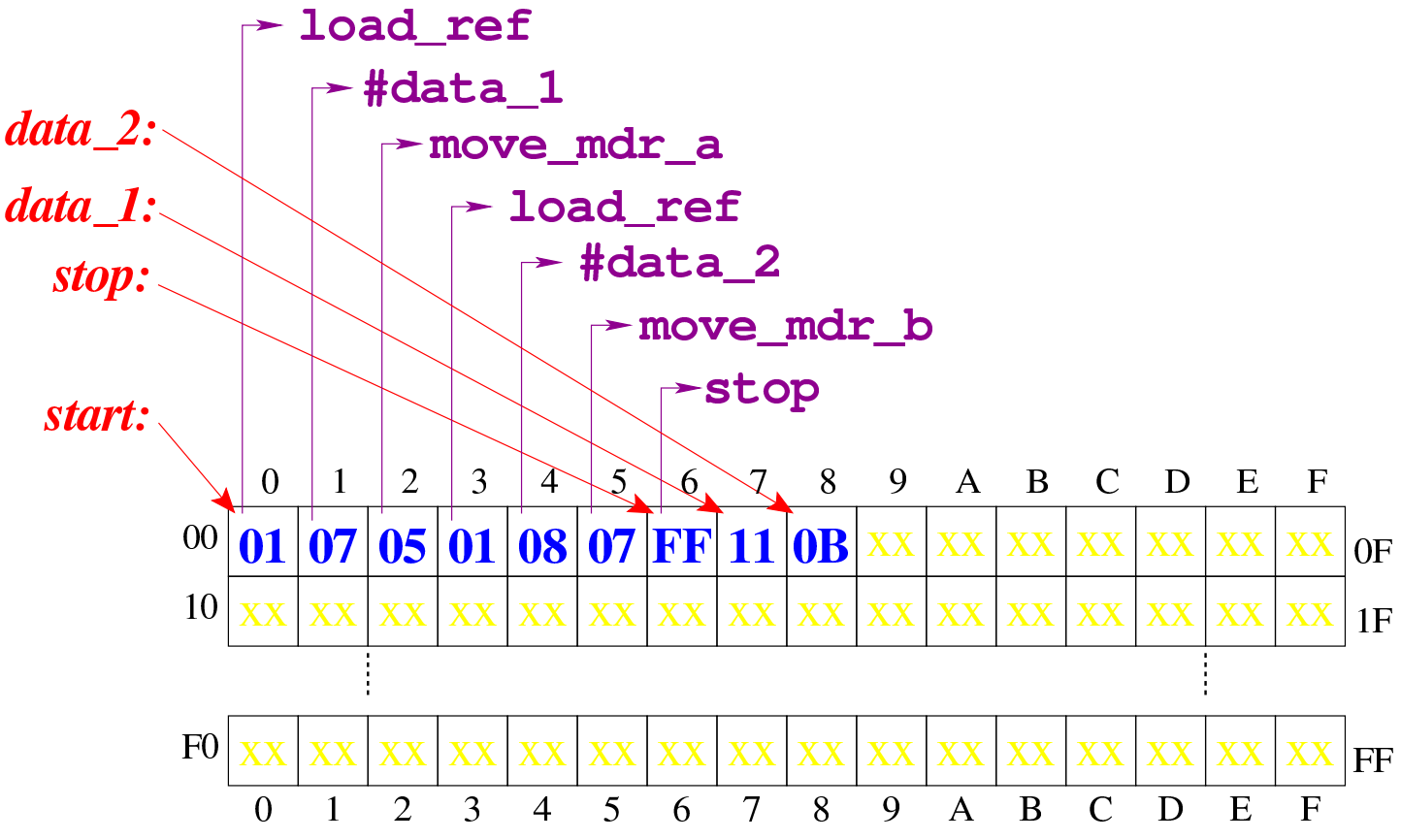
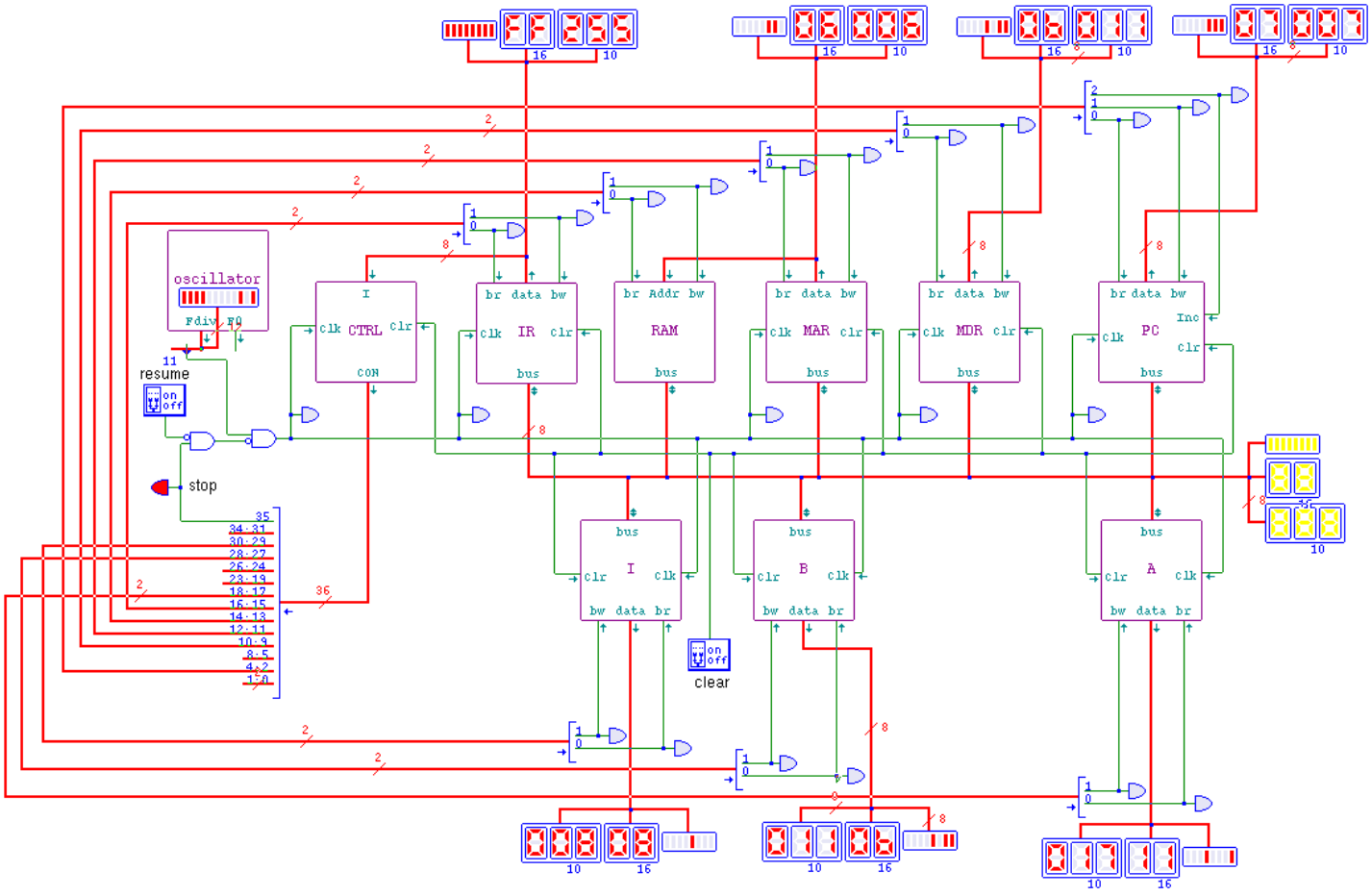


Figure 8.59. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=9qVsCKmxcdk*



At the moment, the simple CPU under development is not able to copy the content of a register to another one, without the mediation of the ***MDR*** register. In other words, there is no **`move_a_b`** instruction and the intended action is obtained instead with two steps: **`move_a_mdr`** and then **`move_mdr_b`**. That is made to keep the project very simple as long as possible.

8.4 Version "A4": ALU

The fourth subversion of the simple CPU receives an ALU (*arithmetic-logic unit*), including the ability to shift and rotate bits, but at the moment there is no management of flags (there are no carry, sign, zero and overflow flags).

Figure 8.60. Simple CPU version "A4".

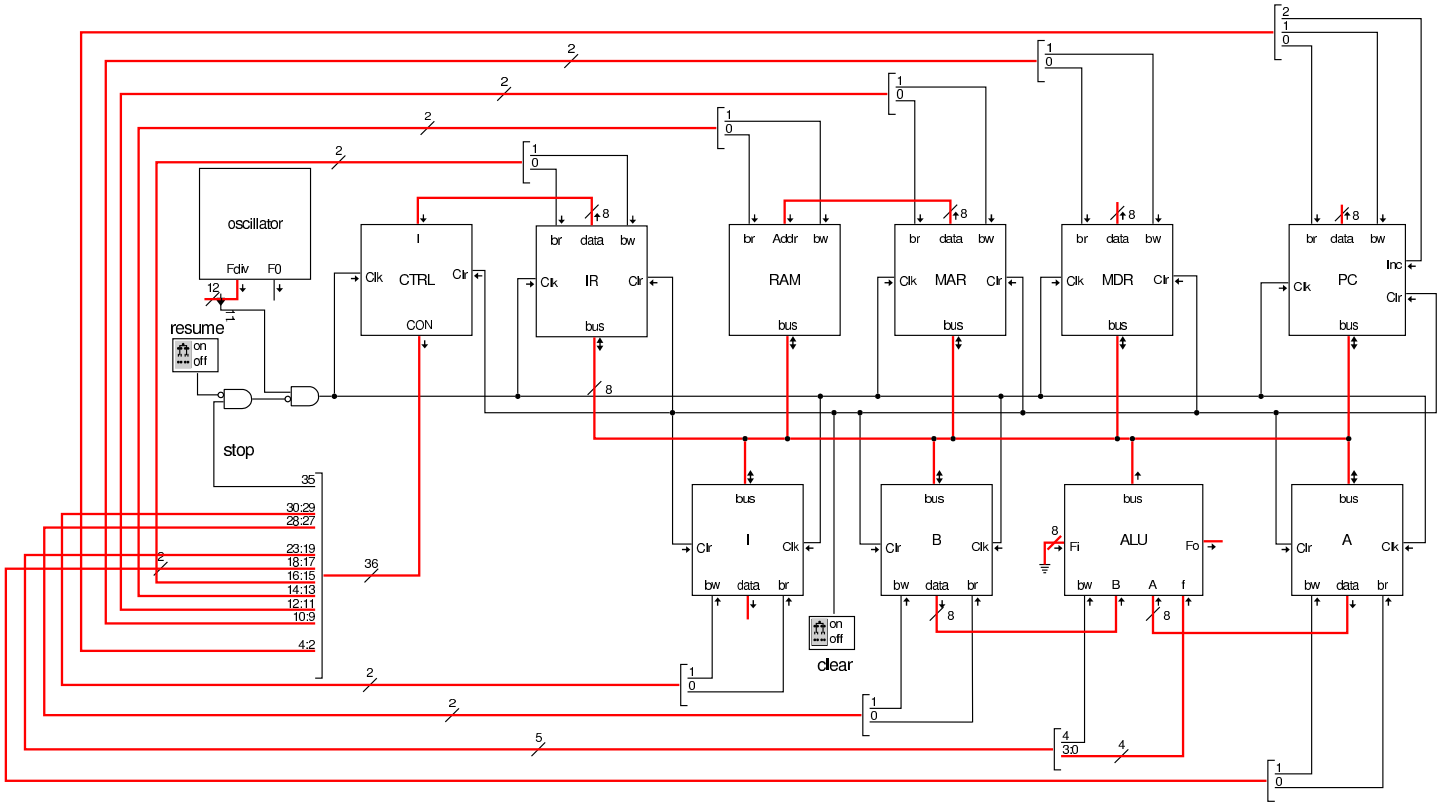


Figure 8.61. The ALU structure: the red lines have an 8-bit size, while the black lines are single bit ones.

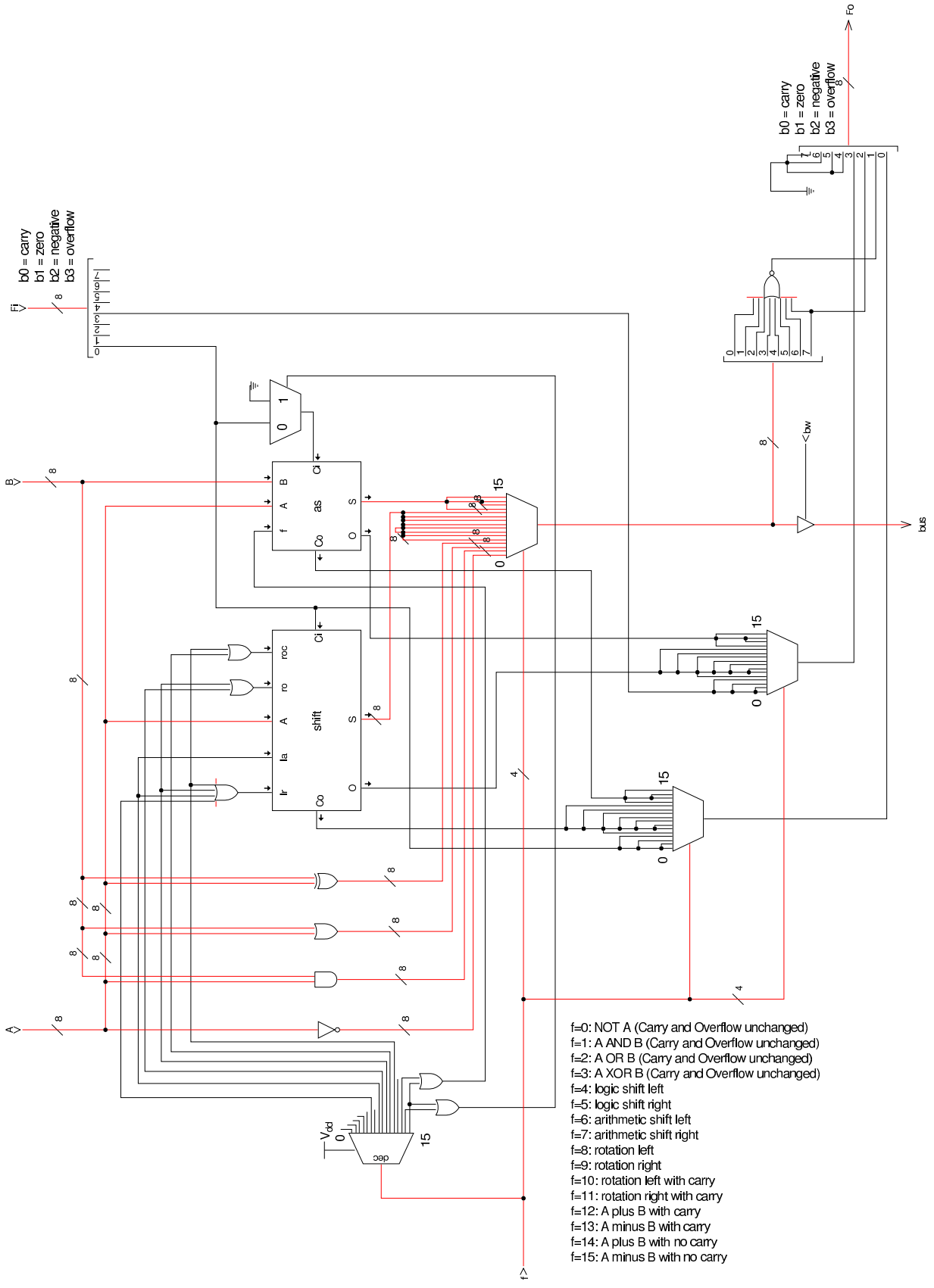


Figure 8.62. Module **`shift`**, responsible to make bit shifts and bit rotations.

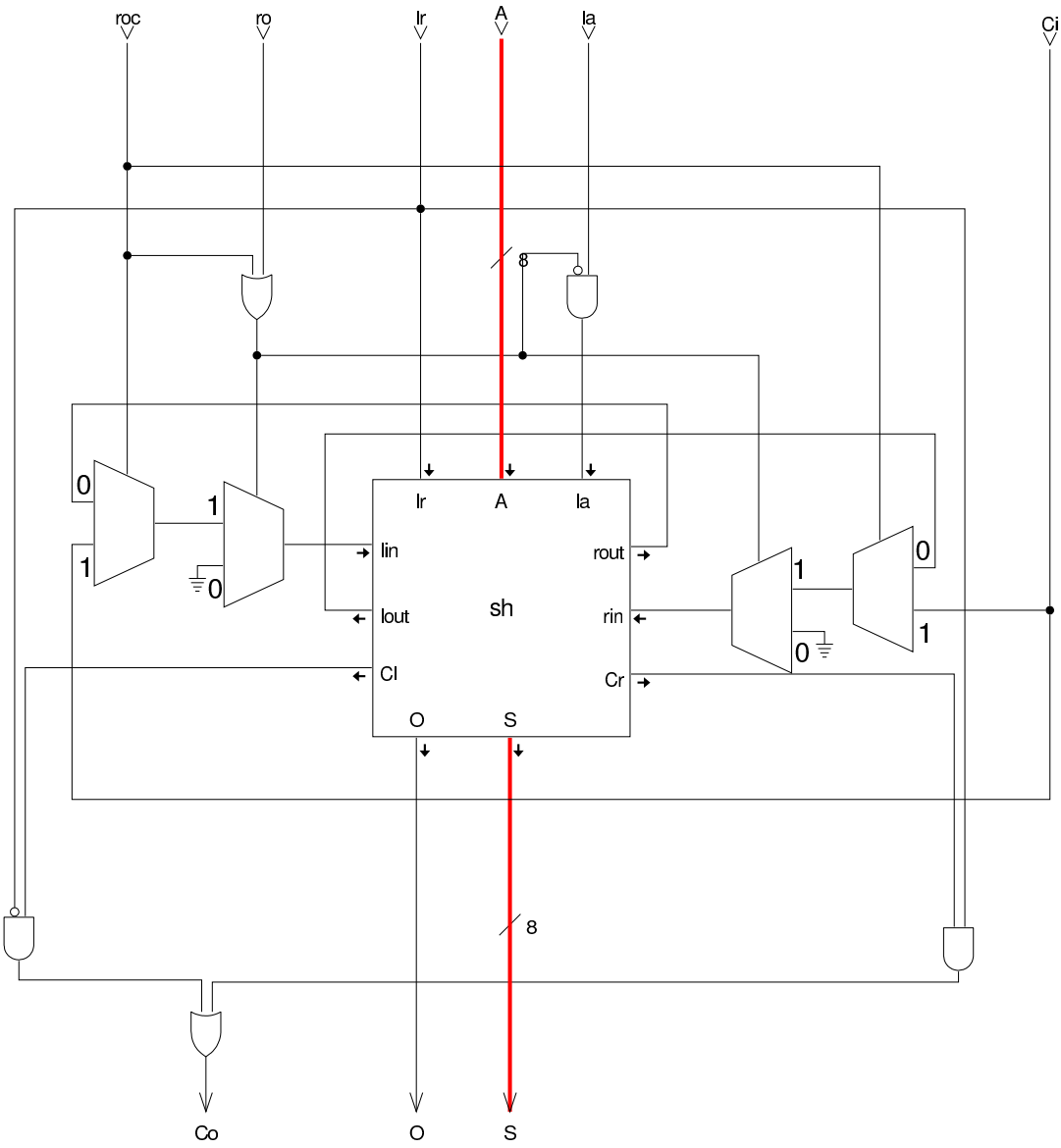


Figure 8.63. Module **sh**, contained inside the **shift** module, to shift the bits.

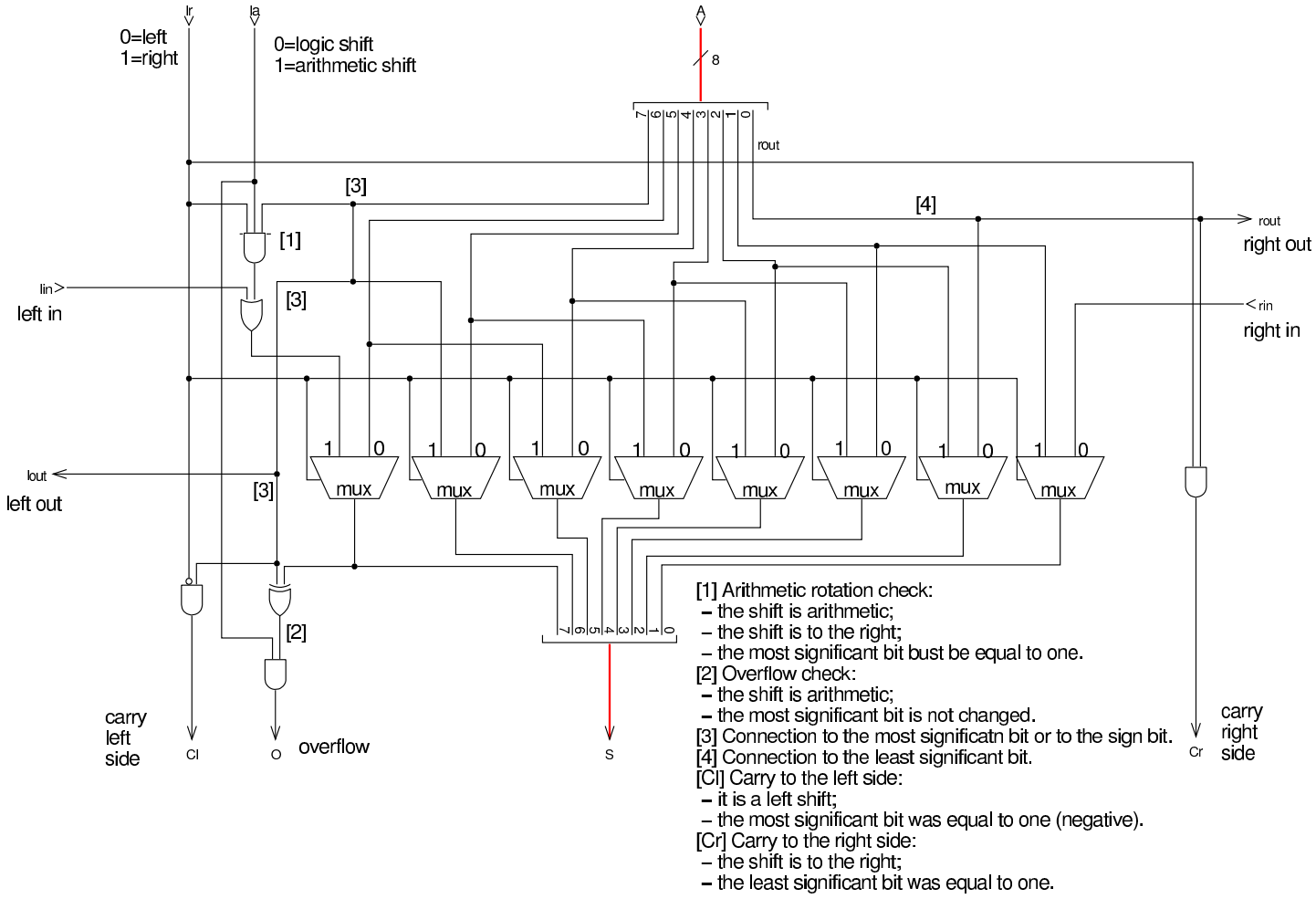
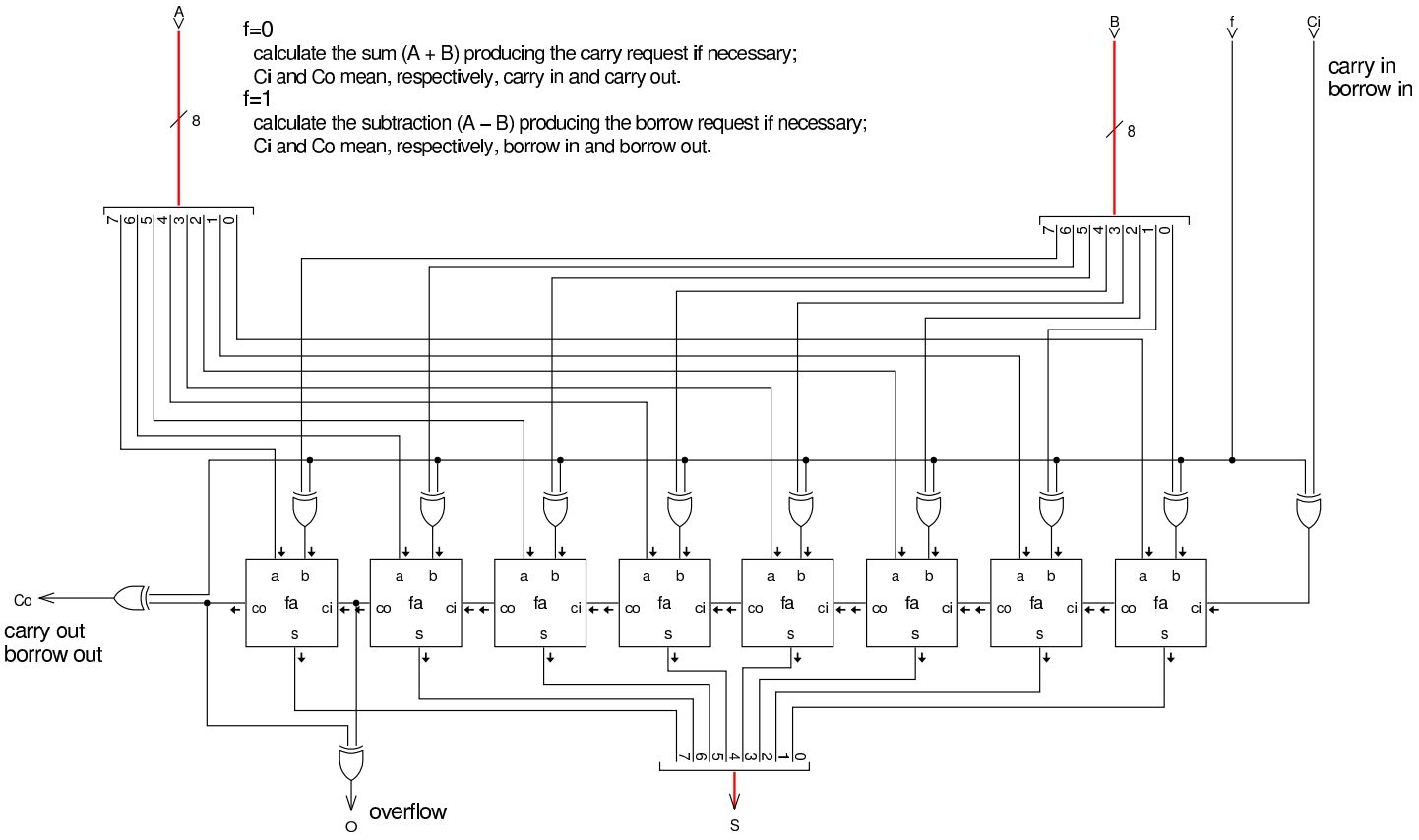


Figure 8.64. Module **as**, which is able to add or subtract the inputs.



Inside the code that describes the microcode word fields, the following fields are added: they are used specifically to manage the ALU. Please notice that the ALU has the ability to write to the bus, but it cannot read from the bus, because the ALU does not receive data from the bus; moreover, the ALU requires to control the function applied to the input data.

```
field alu_f[22:19]={
        not_a=0,
        a_and_b=1,
        a_or_b=2,
        a_xor_b=3,
        logic_shift_left=4,
        logic_shift_right=5,
        arith_shift_left=6,
        arith_shift_right=7,
        rotate_left=8,
        rotate_right=9,
```

```
        rotate_carry_left=10,
        rotate_carry_right=11,
        a_plus_b_carry=12,
        a_minus_b_borrow=13,
        a_plus_b=14,
        a_minus_b=15
};
field alu_bw[23];        // ALU --> bus
field fl_ar[24];         // FL <-- ALU
```

The field ***fl_ar*** can be ignored at the moment, because it will be useful only in the next version of the CPU, when the status flags will be managed. Currently, inside the microcode, there are references to the field ***fl_ar***, because in the next CPU version there will be the register ***FL***, which currently is missing. There are many new opcode for the ALU functions:

```
op not {
  map not : 32;                          // A = NOT A
  +0[7:0]=32;
  operands op_0;
};
op and {
  map and : 33;                          // A = A AND B
  +0[7:0]=33;
  operands op_0;
};
op or {
  map or : 34;                           // A = A OR B
  +0[7:0]=34;
  operands op_0;
};
op xor {
  map xor : 35;                          // A = A OR B
  +0[7:0]=35;
  operands op_0;
};
```

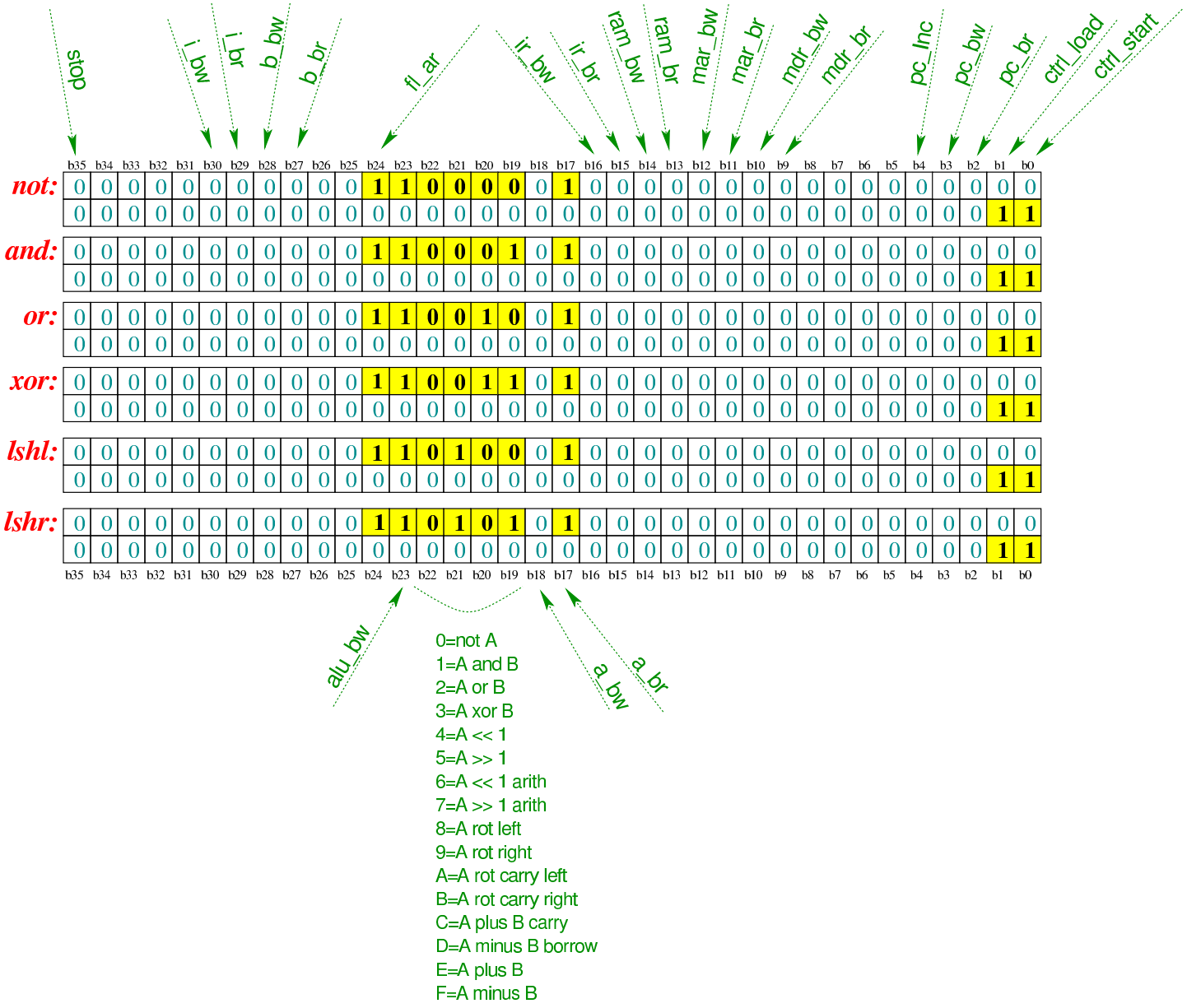
```
op lshl {
  map lshl : 36;                    // A = A << 1
  +0[7:0]=36;
  operands op_0;
};
op lshr {
  map lshr : 37;                    // A = A >> 1
  +0[7:0]=37;
  operands op_0;
};
op ashl {
  map ashl : 38;                    // A = A << 1
  +0[7:0]=38;
  operands op_0;
};
op ashr {
  map ashr : 39;                    // A = +/-A >> 1
  +0[7:0]=39;
  operands op_0;
};
op rotl {
  map rotl : 40;                    // A = A rotate left
  +0[7:0]=40;
  operands op_0;
};
op rotr {
  map rotr : 41;                    // A = A rotate right
  +0[7:0]=41;
  operands op_0;
};
op add {
  map add : 46;                     // A = A + B
  +0[7:0]=46;
  operands op_0;
```

```
};
op sub {
  map sub : 47;                                  // A = A - B
  +0[7:0]=47;
  operands op_0;
};
```

```
begin microcode @ 0
…
not:
    a_br alu_f=not_a alu_bw fl_ar;               // A <-- NOT A
    ctrl_start ctrl_load;                        // CNT <-- 0
//
and:
    a_br alu_f=a_and_b alu_bw fl_ar;             // A <-- A AND B
    ctrl_start ctrl_load;                        // CNT <-- 0
//
or:
    a_br alu_f=a_or_b alu_bw fl_ar;              // A <-- A OR B
    ctrl_start ctrl_load;                        // CNT <-- 0
//
xor:
    a_br alu_f=a_xor_b alu_bw fl_ar;             // A <-- A XOR B
    ctrl_start ctrl_load;                        // CNT <-- 0
//
lshl:
    a_br alu_f=logic_shift_left alu_bw fl_ar;    // A <-- A << 1
    ctrl_start ctrl_load;                        // CNT <-- 0
//
lshr:
    a_br alu_f=logic_shift_right alu_bw fl_ar;   // A <-- A >> 1
    ctrl_start ctrl_load;                        // CNT <-- 0
//
ashl:
    a_br alu_f=arith_shift_left alu_bw fl_ar;    // A <-- A*2
    ctrl_start ctrl_load;                        // CNT <-- 0
```

```
//
ashr:
    a_br alu_f=arith_shift_right alu_bw fl_ar; // A <-- A/2
    ctrl_start ctrl_load;                      // CNT <-- 0
//
rotl:
    a_br alu_f=rotate_left alu_bw fl_ar; // A <-- A rot. left
    ctrl_start ctrl_load;                // CNT <-- 0
//
rotr:
    a_br alu_f=rotate_right alu_bw fl_ar; // A <-- A rot. right
    ctrl_start ctrl_load;                 // CNT <-- 0
//
add:
    a_br alu_f=a_plus_b alu_bw fl_ar;     // A <-- A + B
    ctrl_start ctrl_load;                 // CNT <-- 0
//
sub:
    a_br alu_f=a_minus_b alu_bw fl_ar;    // A <-- A - B
    ctrl_start ctrl_load;                 // CNT <-- 0
…
end
```

Figure 8.68. Match between the contents of the memory that represents the microcode (the couple of memory banks **`m1`** and **`m2`** inside the control unit) and the microcode description itself.



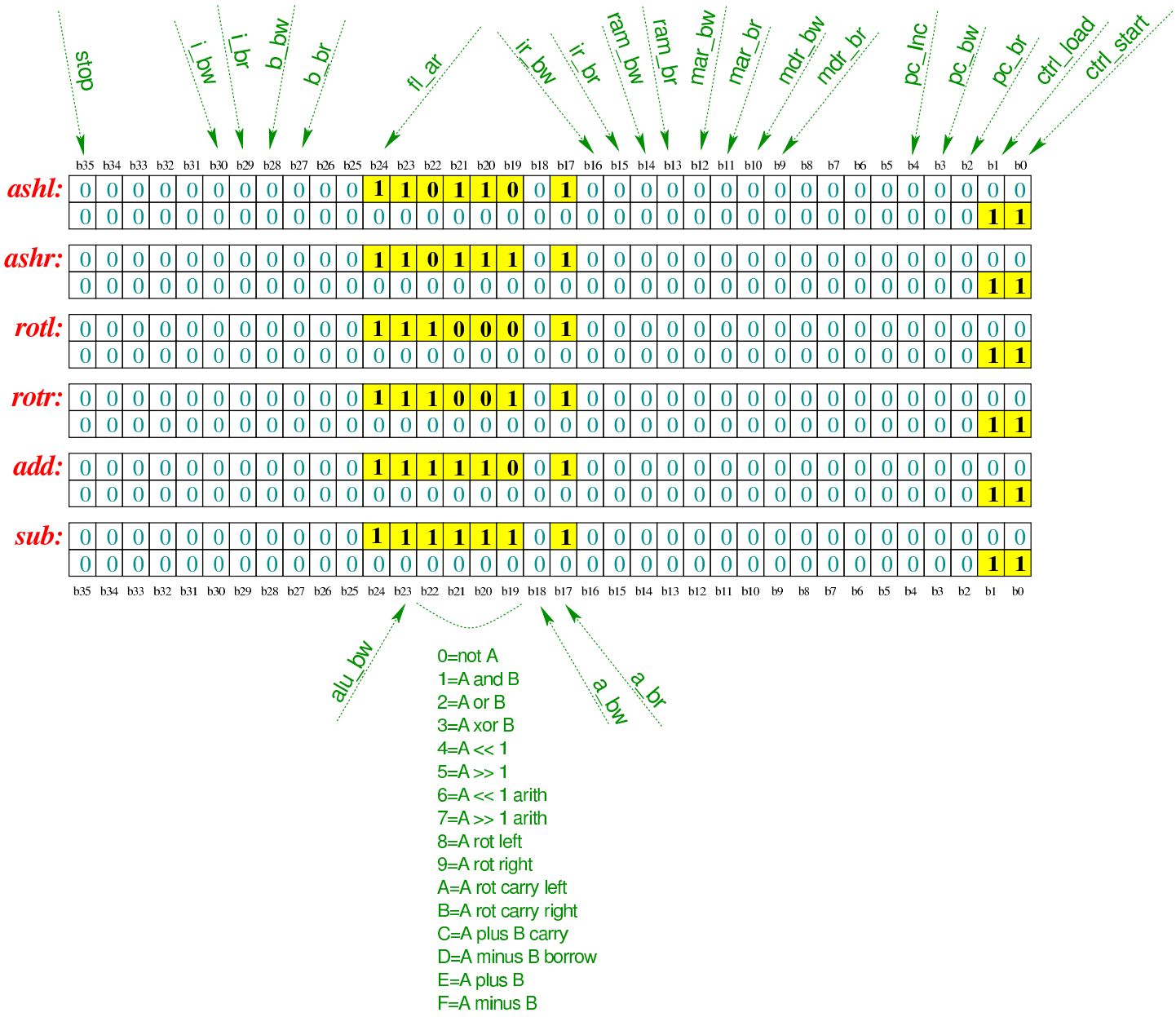


Table 8.69. The new macroinstructions for the current CPU version. Some C language notation is used to simplify the description.

Syntax	Description
`not`	***A*** = ~***A*** One's complement of ***A***.
`and`	***A*** = ***A*** & ***B*** It is assigned to ***A*** the result of ***A*** AND ***B***, bit per bit.
`or`	***A*** = ***A*** | ***B*** It is assigned to ***A*** the result of ***A*** OR ***B***, bit per bit.

Syntax	Description
`xor`	$A = A \text{ ^ } B$ It is assigned to A the result of A XOR B, bit per bit.
`lshl`	$A = A << 1$ It is assigned to A the logic shift left of one bit from the previous value of A.
`lshr`	$A = A >> 1$ It is assigned to A the logic shift right of one bit from the previous value of A.
`ashl`	$A = A << 1$ It is assigned to A the arithmetic shift left of one bit from the previous value of A (it is the same as the logic shift left).
`ashr`	$A = A >> 1$ It is assigned to A the arithmetic shift right of one bit from the previous value of A (the sign is kept the same).
`rotl`	It is assigned to A the result of one bit rotation left from the original value of A.
`rotr`	It is assigned to A the result of one bit rotation right from the original value of A.
`add`	$A = A + B$ it is assigned to A the result of the sum of A and B, without considering the previous carry.
`sub`	$A = A - B$ it is assigned to A the result of the subtraction of A and B, without considering the previous borrow.

The following sections offer various examples, where the use of all the new instructions is shown.

8.4.1 Instruction "not"

«

Listing 8.70. Macrocode example to experiment the **not** instruction: a value is loaded from the memory to the ***A*** register, then the one's complement is calculated and put into the ***A*** register. The complete source should be available at *attachments/xcpu/xcpu-a4-not.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        not
stop:
        stop
data_1:
        .byte 17
end
```

Figure 8.71. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

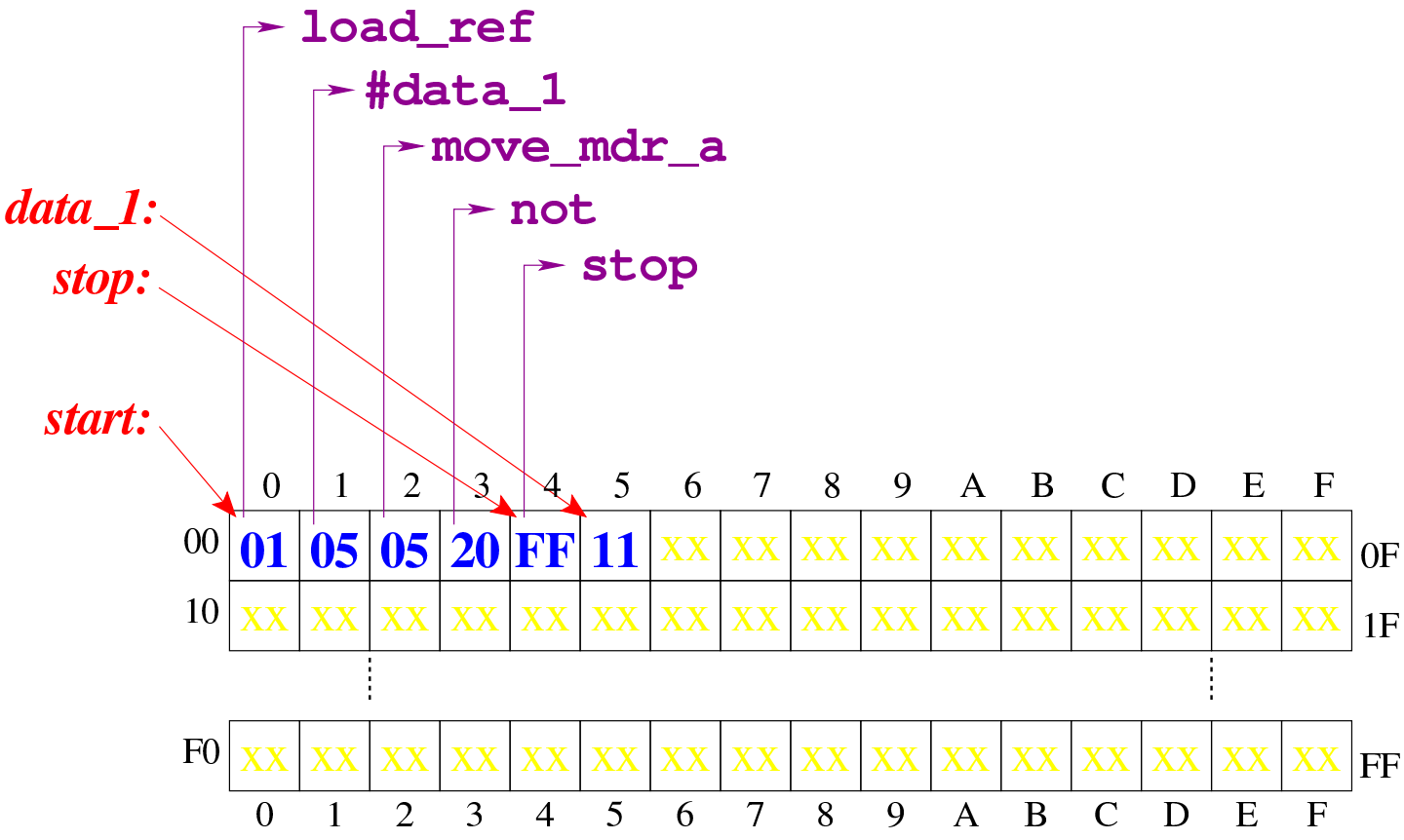
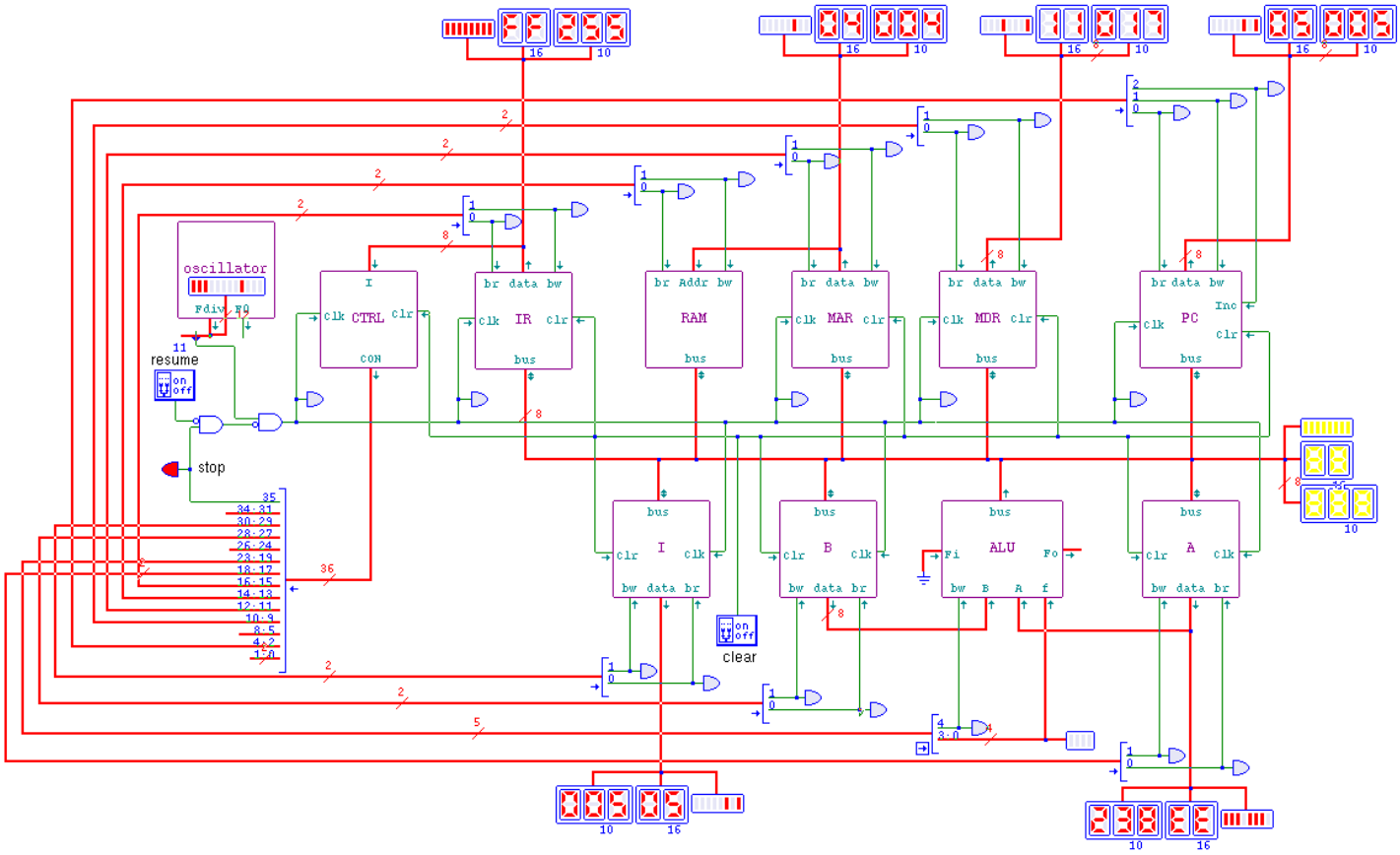


Figure 8.72. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=x5Vnhd72vh728*



8.4.2 Instruction "and"

Listing 8.73. Macrocode example to experiment the **and** instruction: two values are loaded from the memory into the registers ***A*** and ***B***, then the binary AND is calculated and saved inside the ***A*** register. The complete source should be available at *attachments/xcpu/xcpu-a4-and.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        load_ref #data_2
        move_mdr_b
        and
stop:
        stop
```

```
data_1:
        .byte 17
data_2:
        .byte 11
end
```

Figure 8.74. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

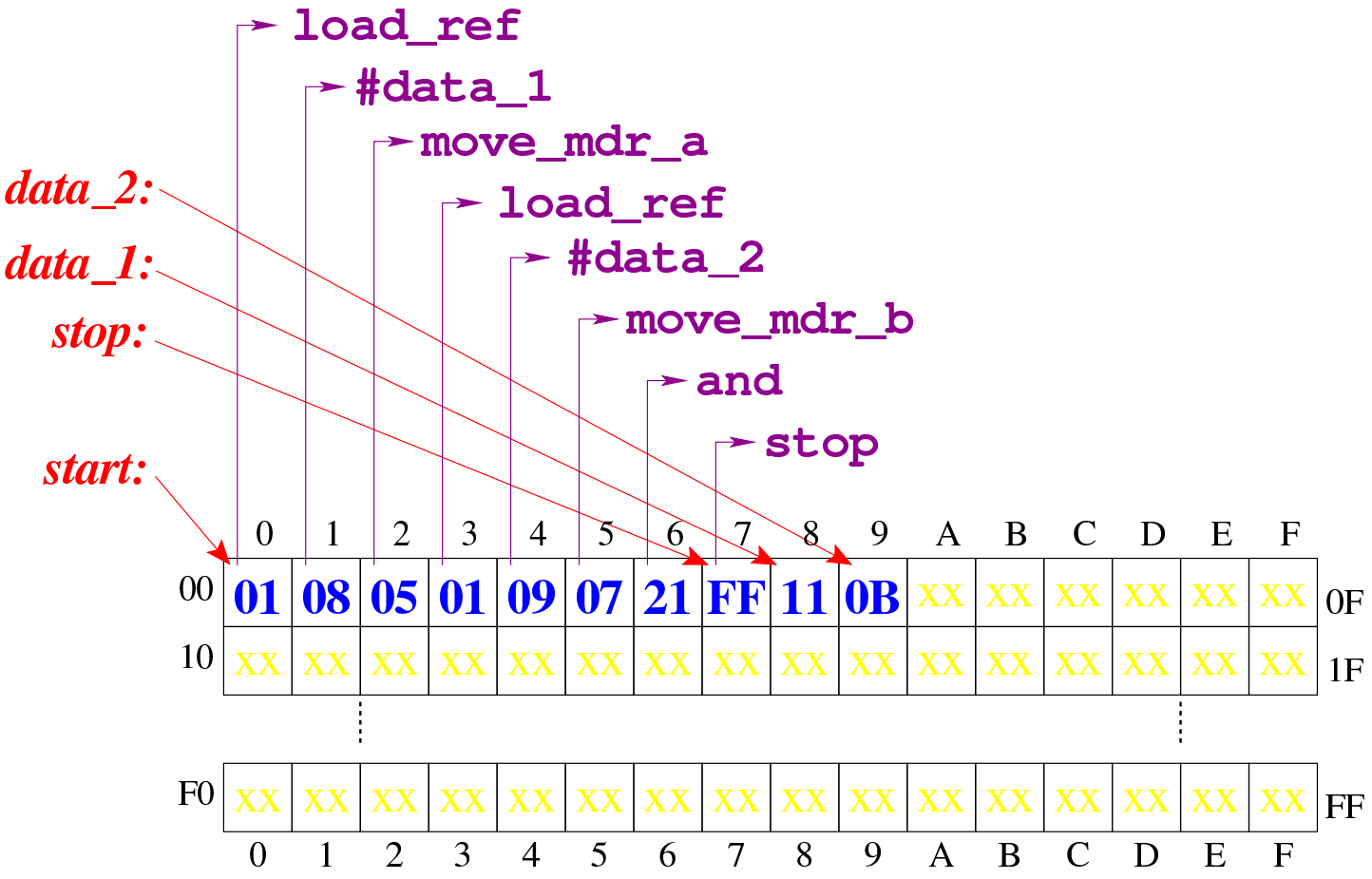
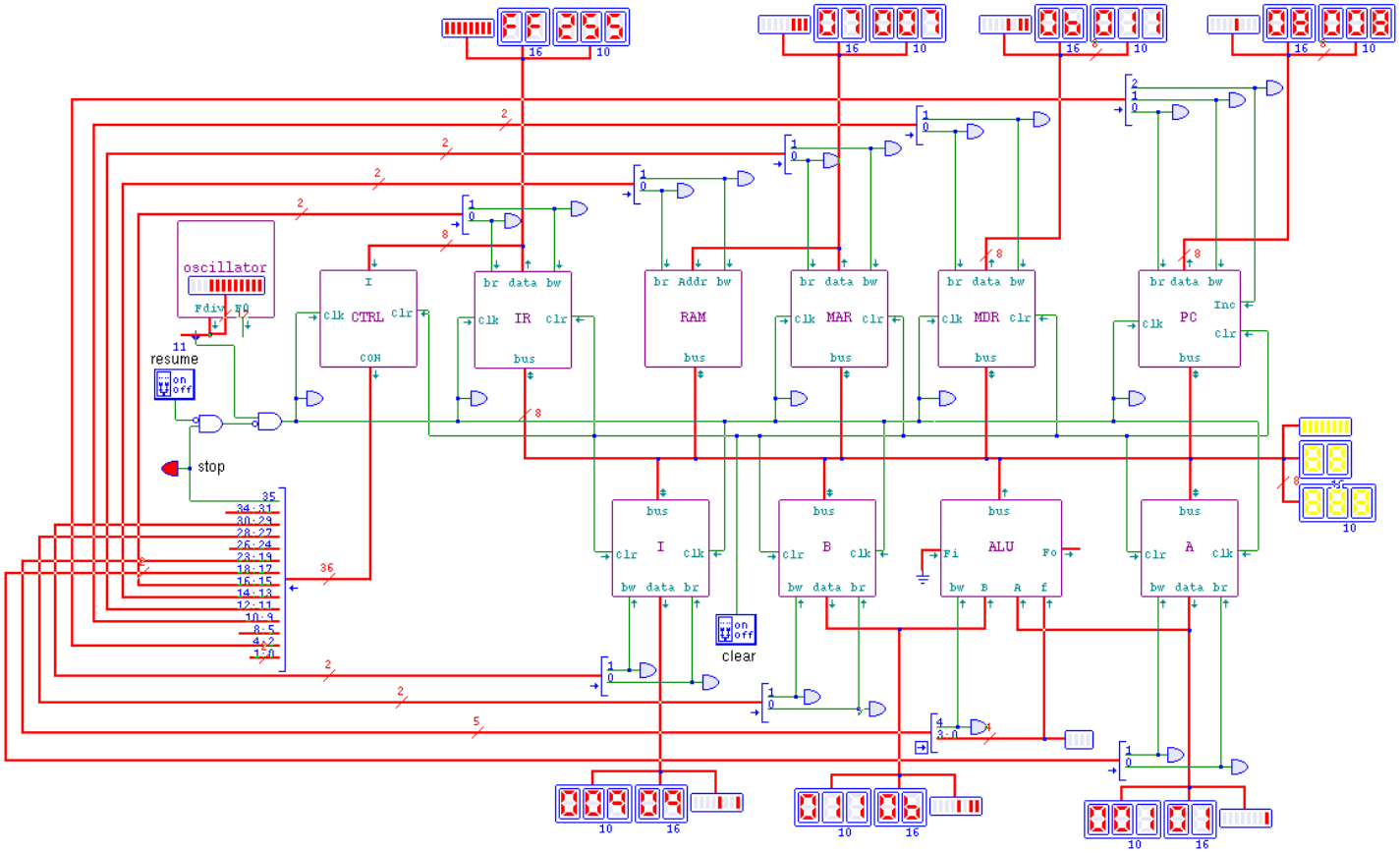


Figure 8.75. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=2ra7SHxBvYY*



8.4.3 Instruction "or"

«

Listing 8.76. Macrocode example to experiment the **`or`** instruction: two values are loaded from the memory into the registers ***A*** and ***B***, then the binary OR is calculated and saved inside the ***A*** register. The complete source should be available at *attachments/xcpu/xcpu-a4-or.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        load_ref #data_2
        move_mdr_b
        or
stop:
        stop
```

```
data_1:
        .byte 17
data_2:
        .byte 11
end
```

Figure 8.77. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

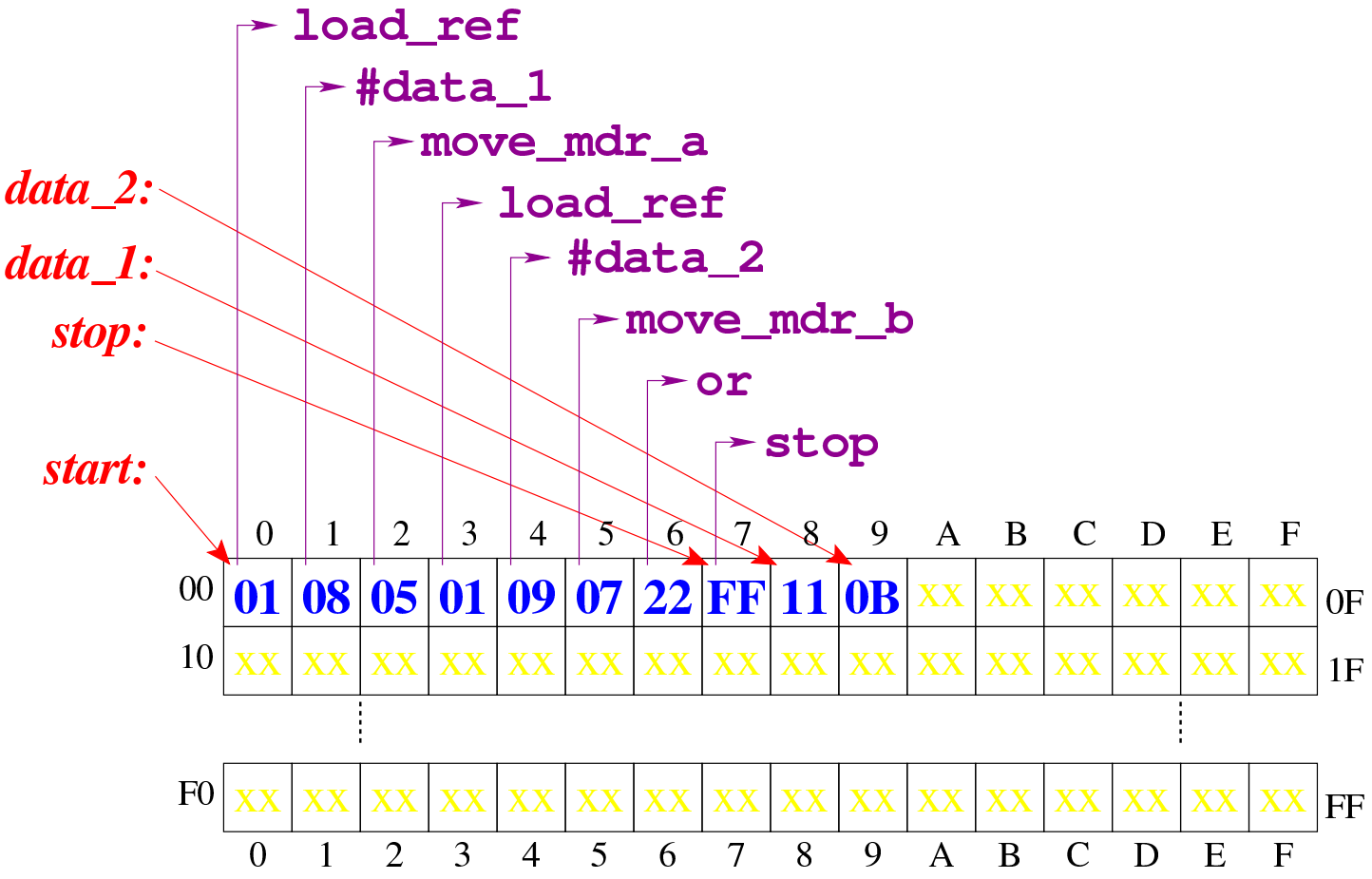
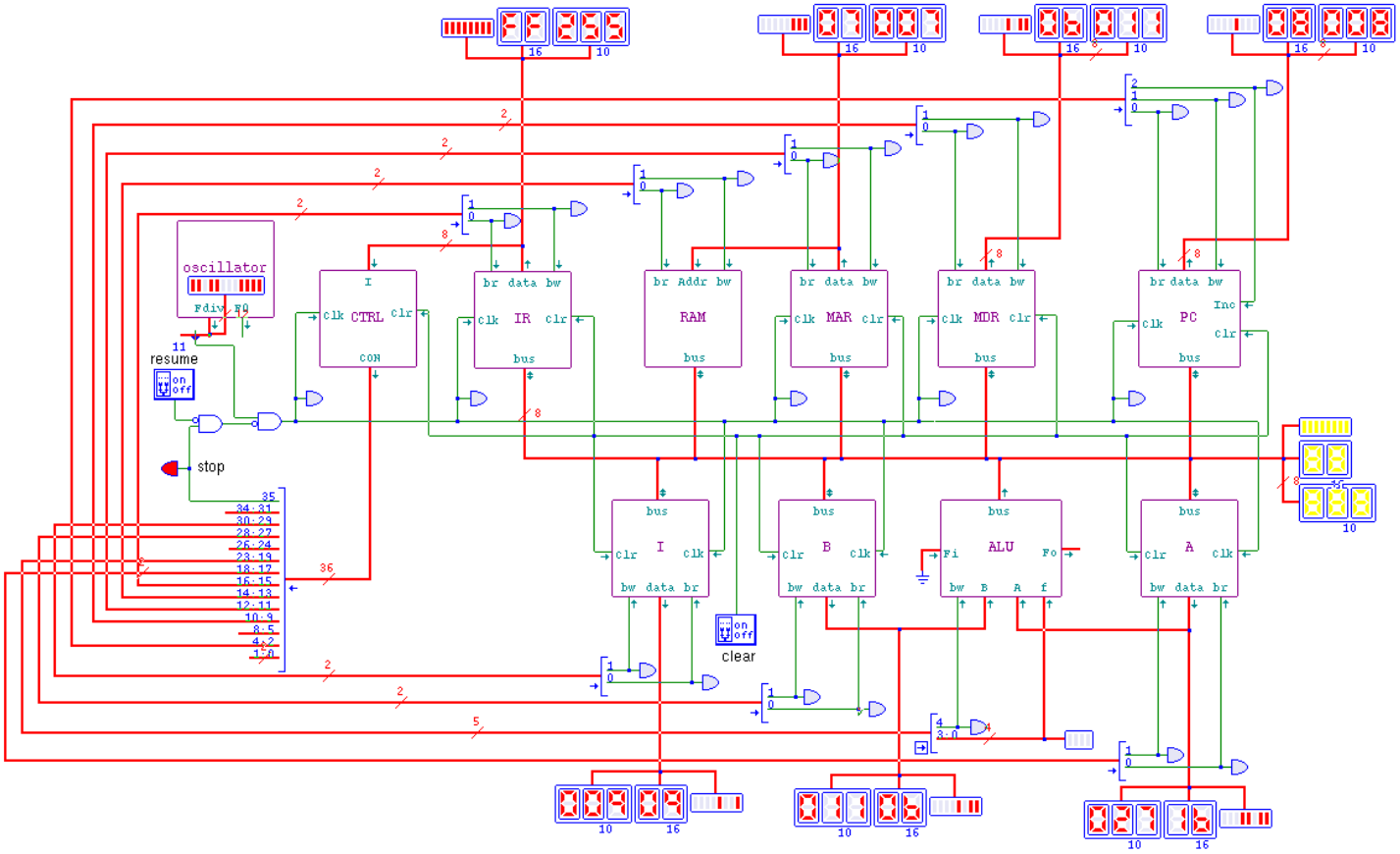


Figure 8.78. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=7E-2uA6fVoY*



8.4.4 Instruction "xor"

Listing 8.79. Macrocode example to experiment the **`or`** instruction: two values are loaded from the memory into the registers ***A*** and ***B***, then the binary XOR is calculated and saved inside the ***A*** register. The complete source should be available at *attachments/xcpu/xcpu-a4-xor.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        load_ref #data_2
        move_mdr_b
        xor
stop:
        stop
```

```
data_1:
        .byte 17
data_2:
        .byte 11
end
```

Figure 8.80. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

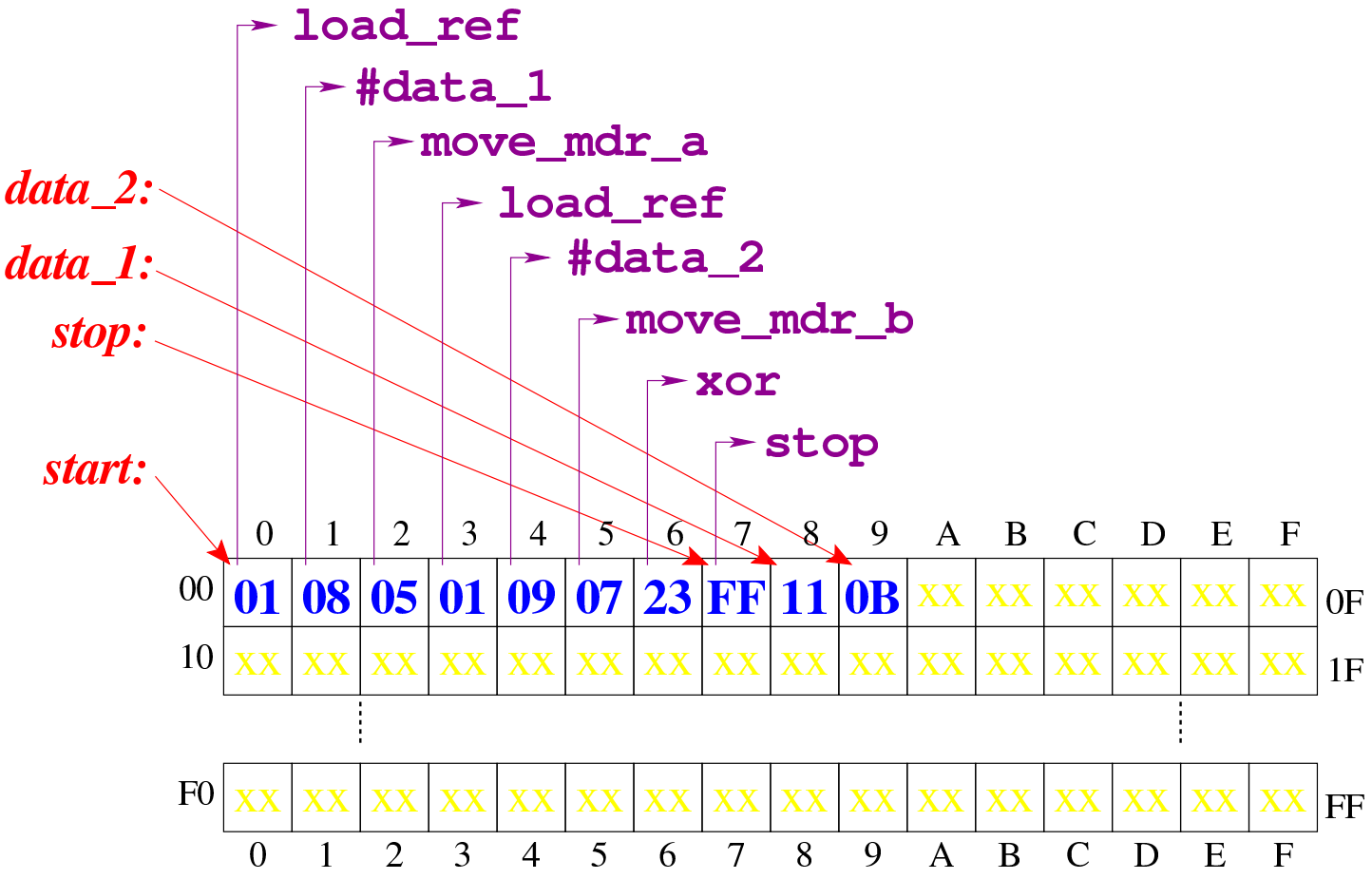
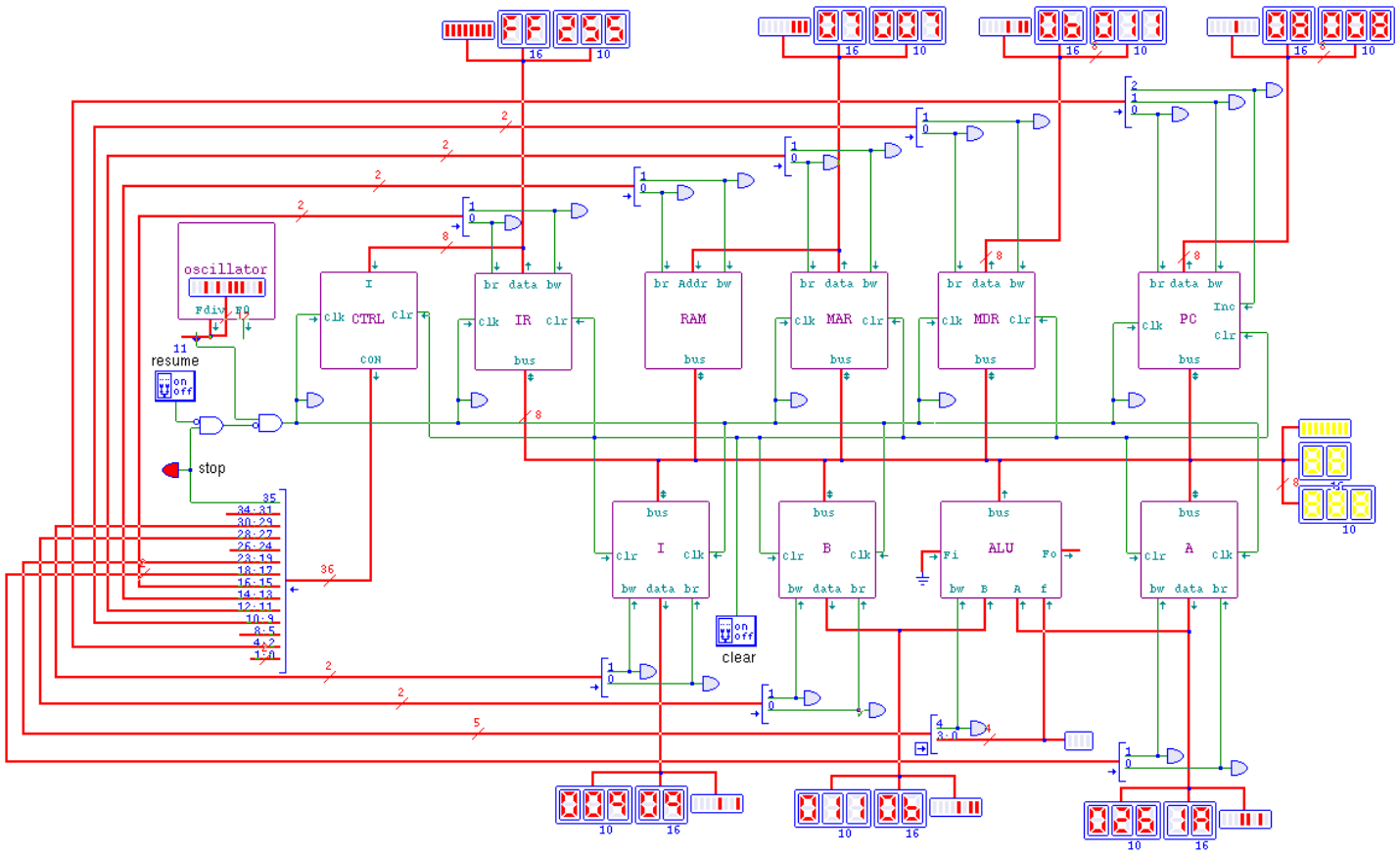


Figure 8.81. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=TuzknbyeabQ*



8.4.5 Instruction "lshl" and "lshr"

Listing 8.82. Macrocode example to experiment the logic shift instructions: a value is loaded from the memory to the ***A*** register and a left shift is done, updating the value of ***A***. Then the result is copied into ***B***, before another value is loaded to calculate the right shift. The complete source file should be available at *attachments/xcpu/xcpu-a4-lsh.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        lshl
        move_a_mdr
        move_mdr_b
        load_ref #data_1
```

```
        move_mdr_a
        lshr
stop:
        stop
data_1:
        .byte 17
end
```

Figure 8.83. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

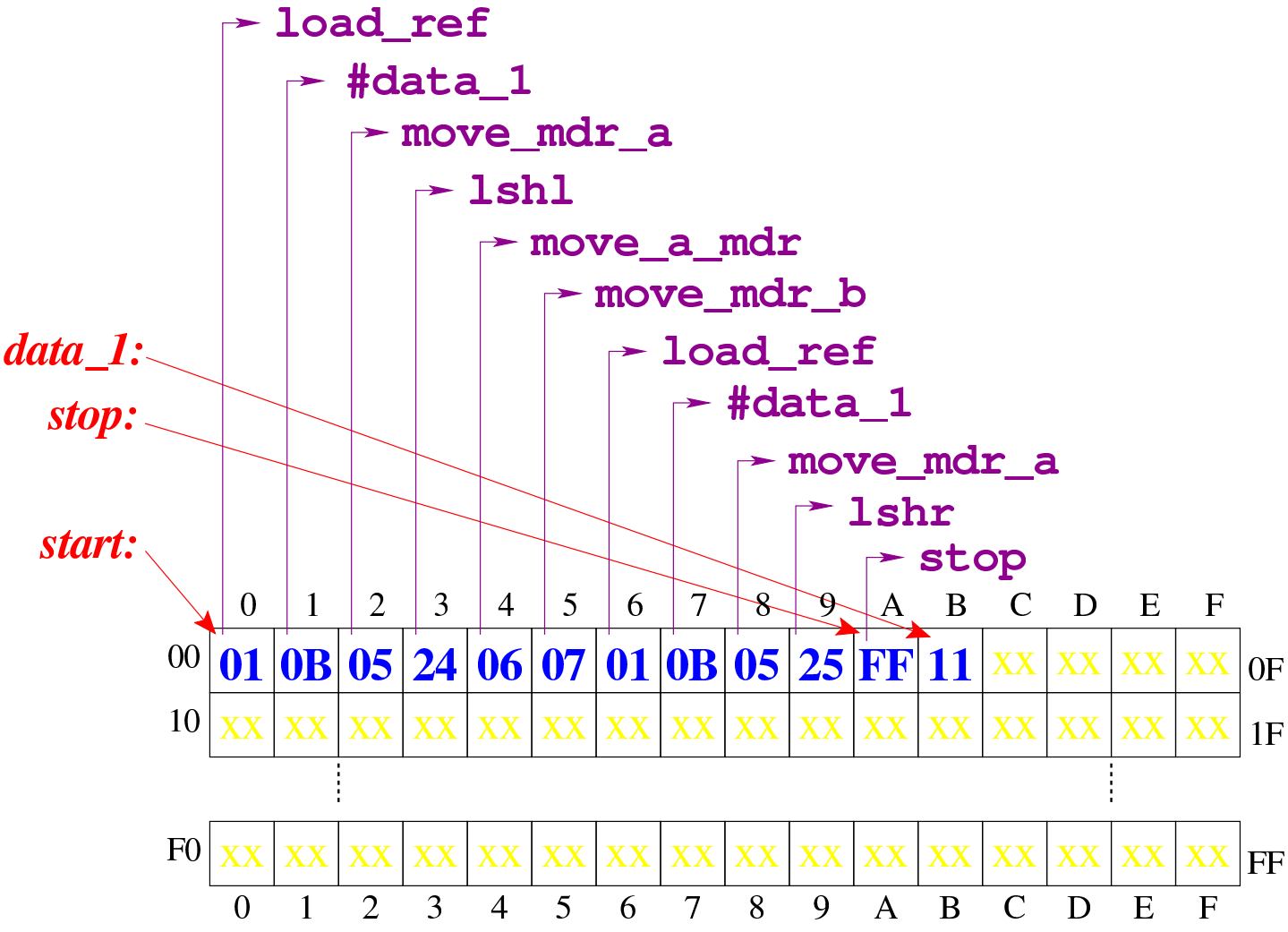
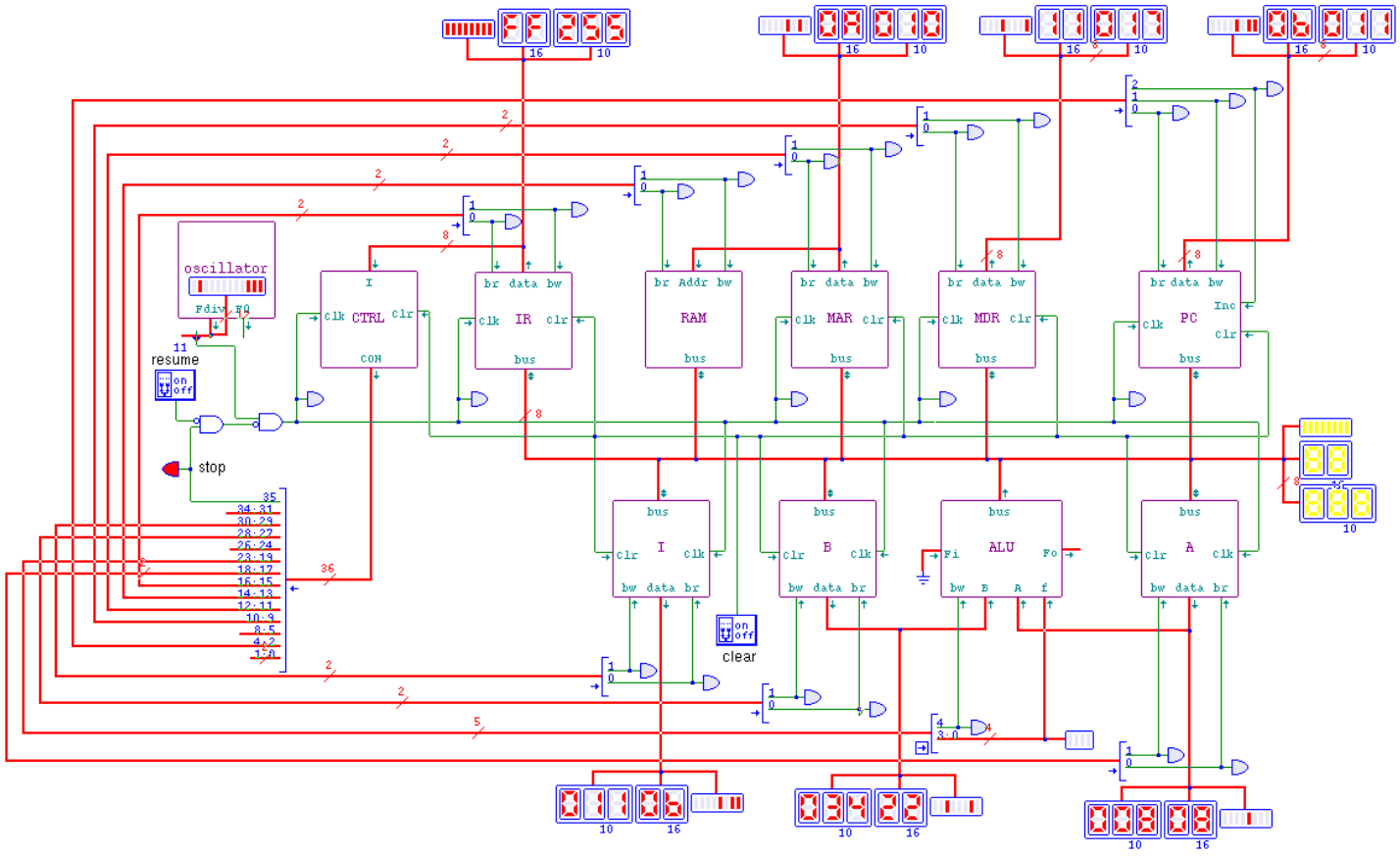


Figure 8.84. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=pkRfWYqGeB4*



8.4.6 Instruction "ashl" and "ashr"

«

Listing 8.85. Macrocode example to experiment the arithmetic shift instructions: a value is loaded from the memory to the ***A*** register and a left shift is done, updating the value of ***A***. Then the result is copied into ***B***, before another value is loaded to calculate the right shift. The complete source file should be available at *attachments/xcpu/xcpu-a4-ash.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        ashl
        move_a_mdr
        move_mdr_b
        load_ref #data_1
```

```
        move_mdr_a
        ashr
stop:
        stop
data_1:
        .byte 143
end
```

Figure 8.86. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

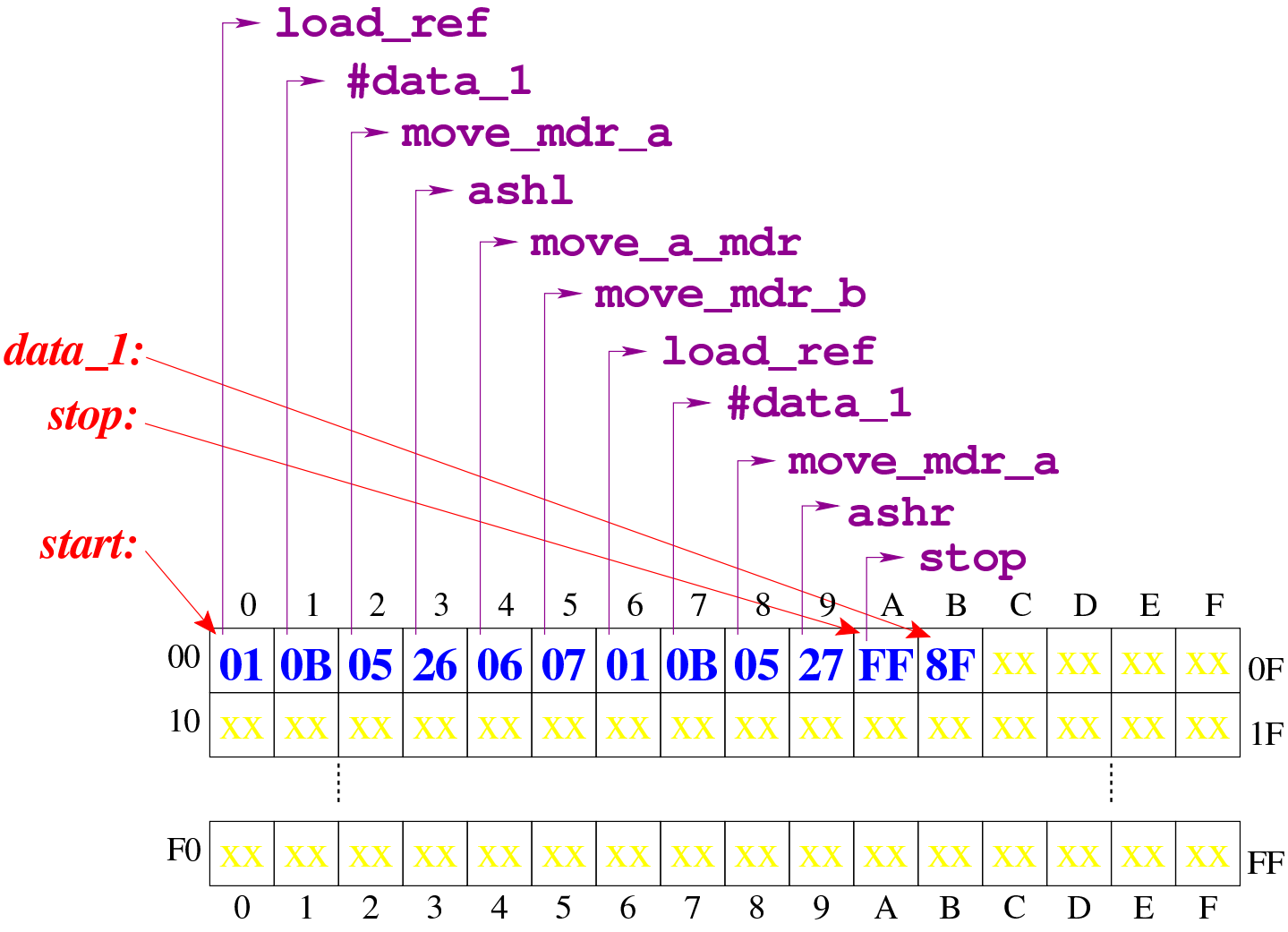
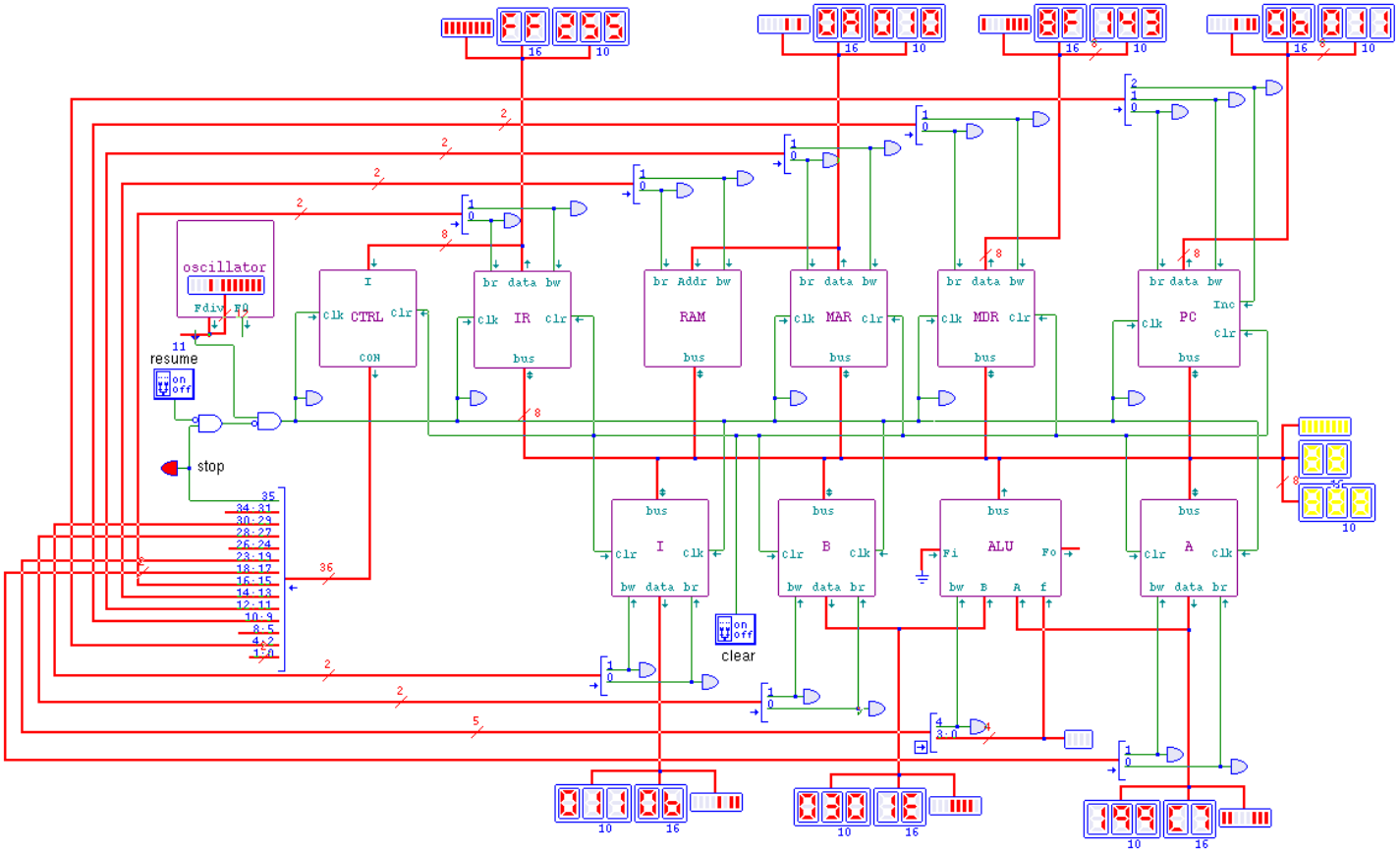


Figure 8.87. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=3rvR1WvWd1k*



8.4.7 Instruction "rotl" and "rotr"

Listing 8.88. Macrocode example to experiment the rotation instructions: a value is loaded from the memory to the ***A*** register and a left rotation is done, updating the value of ***A***. Then the result is copied into ***B***, before another value is loaded to calculate the right rotation. The complete source file should be available at *attachments/xcpu/xcpu-a4-rot.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        rotl
        move_a_mdr
        move_mdr_b
        load_ref #data_1
```

```
        move_mdr_a
        rotr
stop:
        stop
data_1:
        .byte 17
end
```

Figure 8.89. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

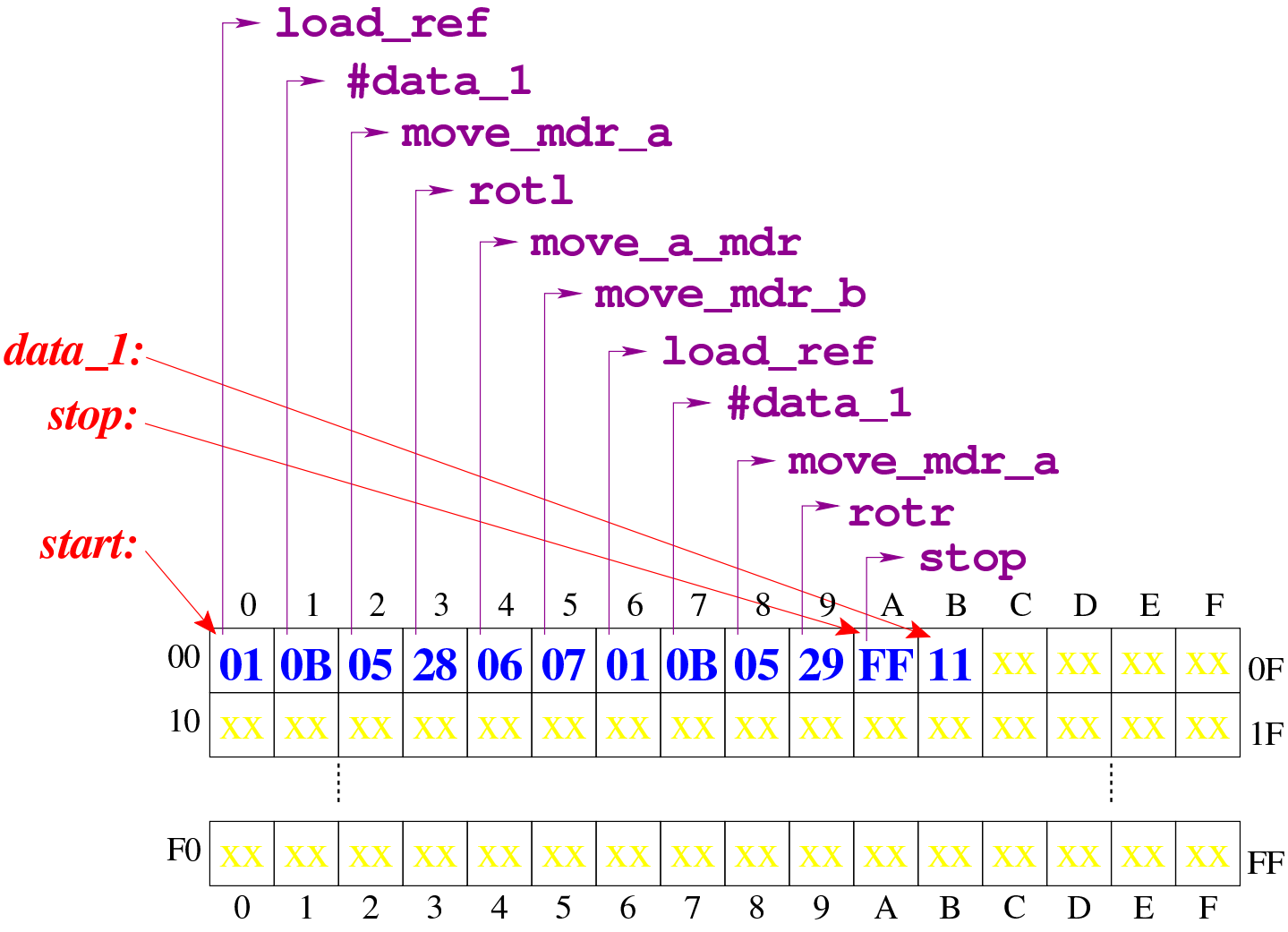
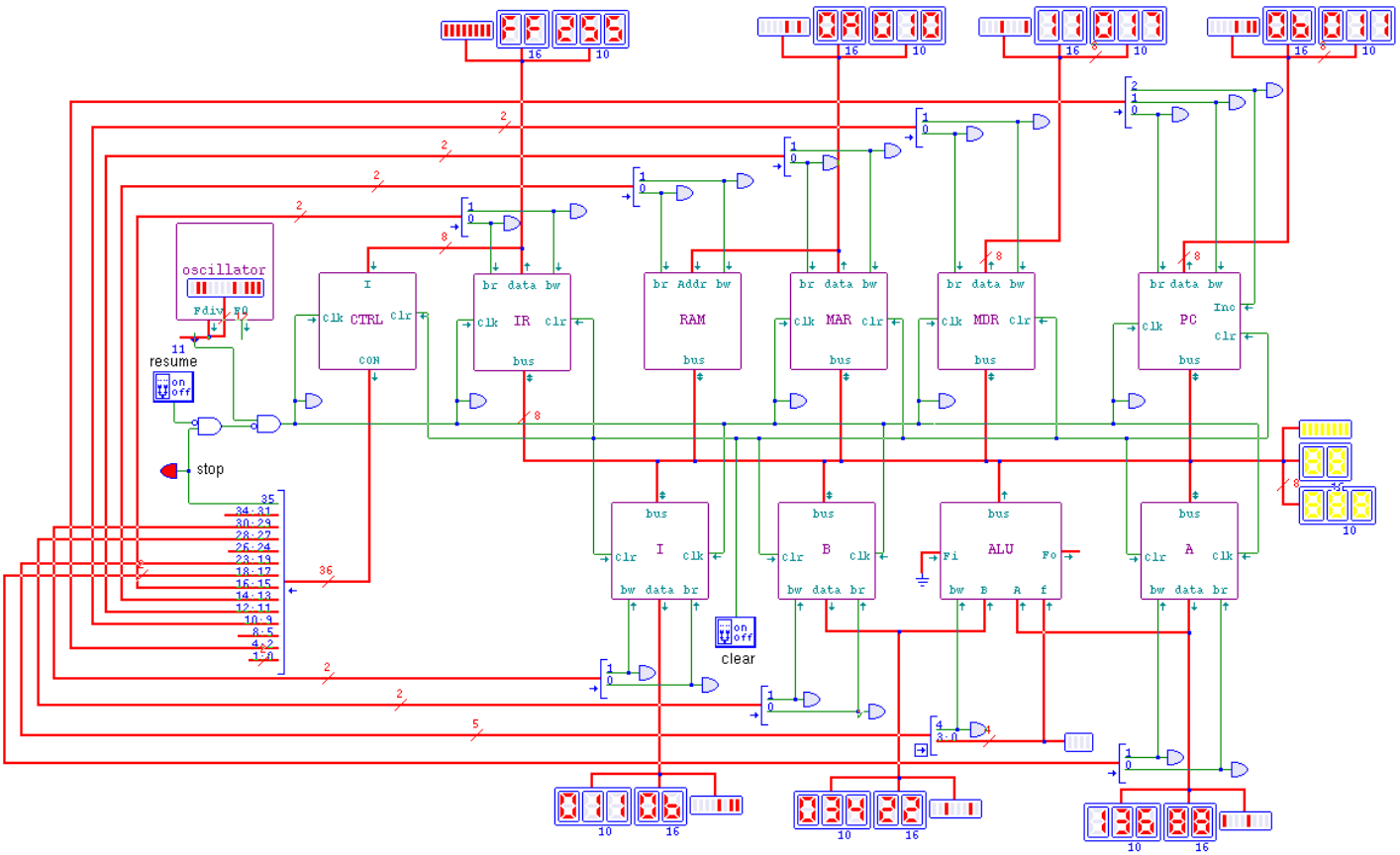


Figure 8.90. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=KCi8n6bnLQo*



8.4.8 Instruction "add"

Listing 8.91. Macrocode example to experiment the addition instruction: two values are loaded from the memory into the registers ***A*** and ***B***, then the addition is calculated and saved inside the ***A*** register. The complete source should be available at *attachments/xcpu/xcpu-a4-add.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        load_ref #data_2
        move_mdr_b
        add
stop:
        stop
```

```
data_1:
        .byte 17
data_2:
        .byte 11
end
```

Figure 8.92. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

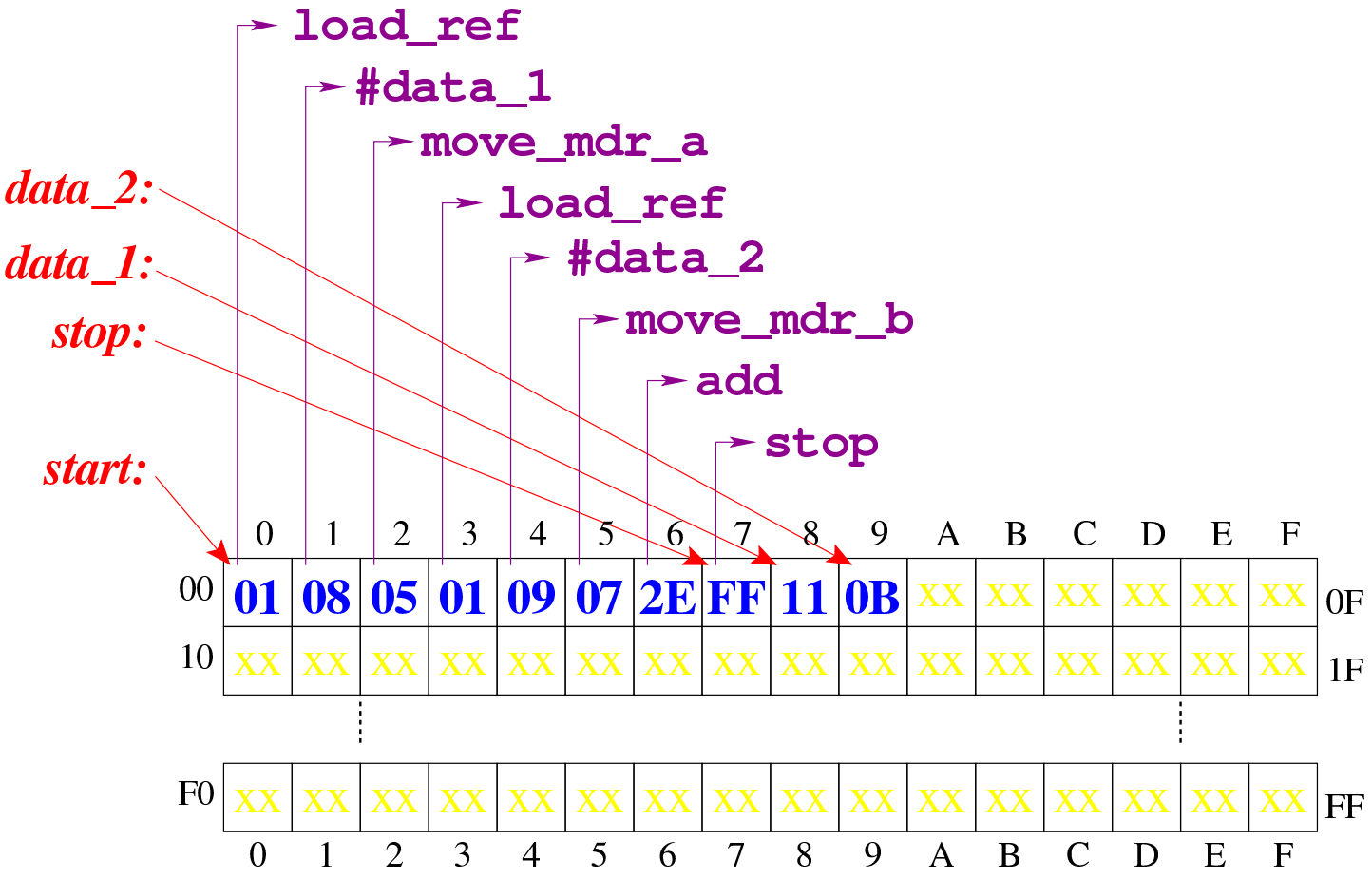
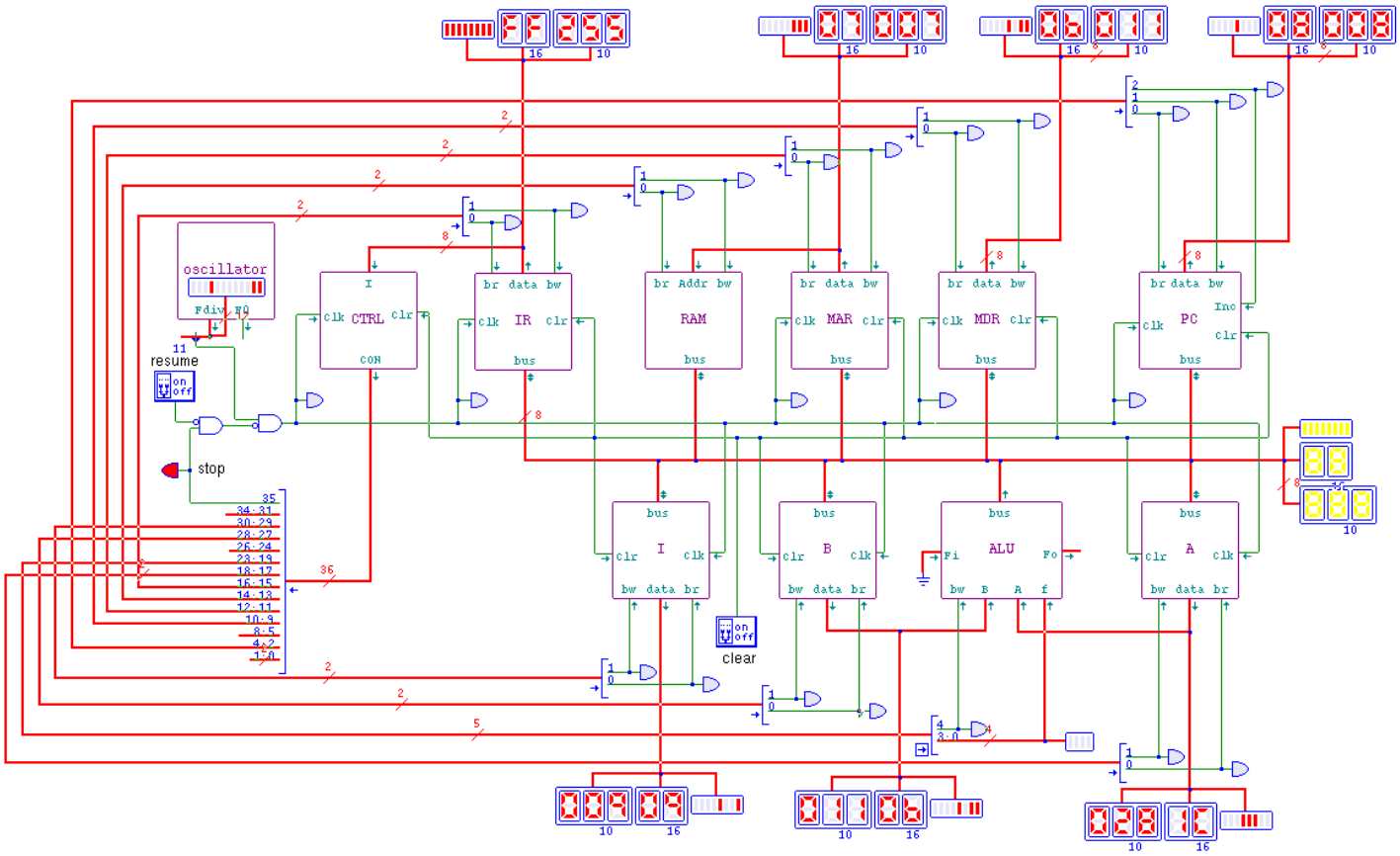


Figure 8.93. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=QQJwz2yVwA8*



8.4.9 Instruction "sub"

Listing 8.94. Macrocode example to experiment the subtraction instruction: two values are loaded from the memory into the registers ***A*** and ***B***, then the subtraction (***A*** − ***B***) is calculated and saved inside the ***A*** register. The complete source should be available at *attachments/xcpu/xcpu-a4-sub.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        load_ref #data_2
        move_mdr_b
        sub
stop:
        stop
```

```
data_1:
        .byte 17
data_2:
        .byte 11
end
```

Figure 8.95. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

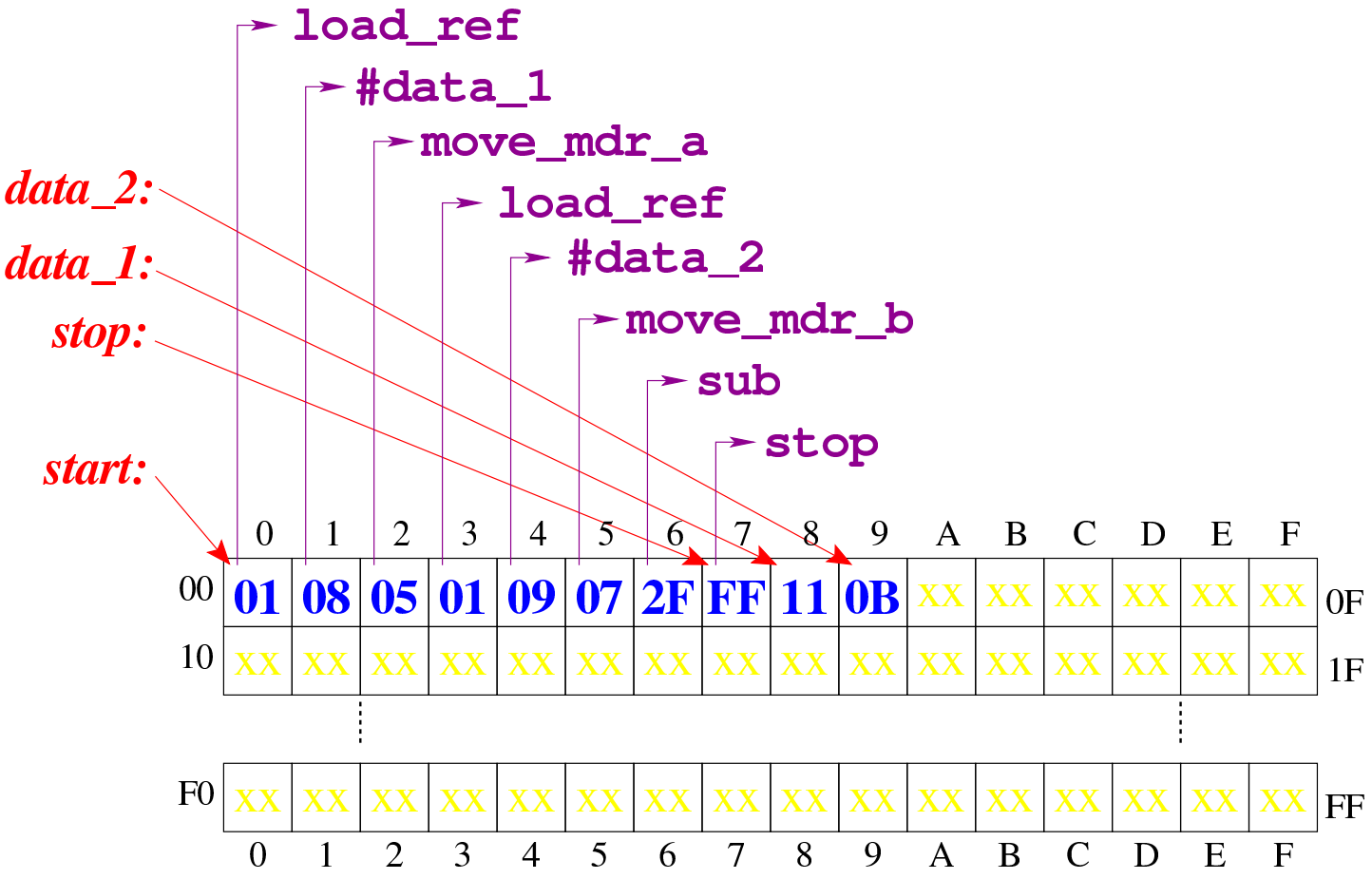
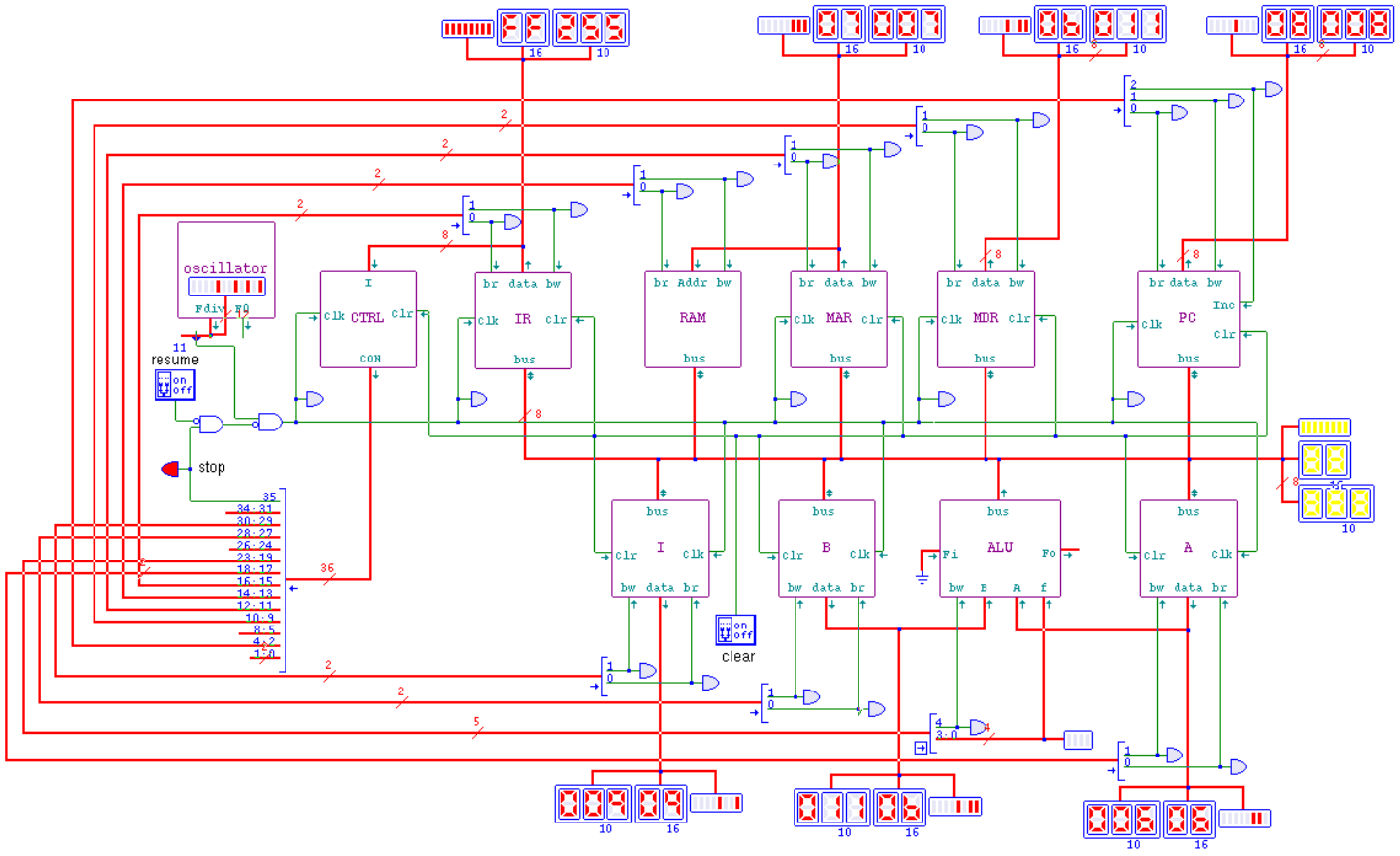


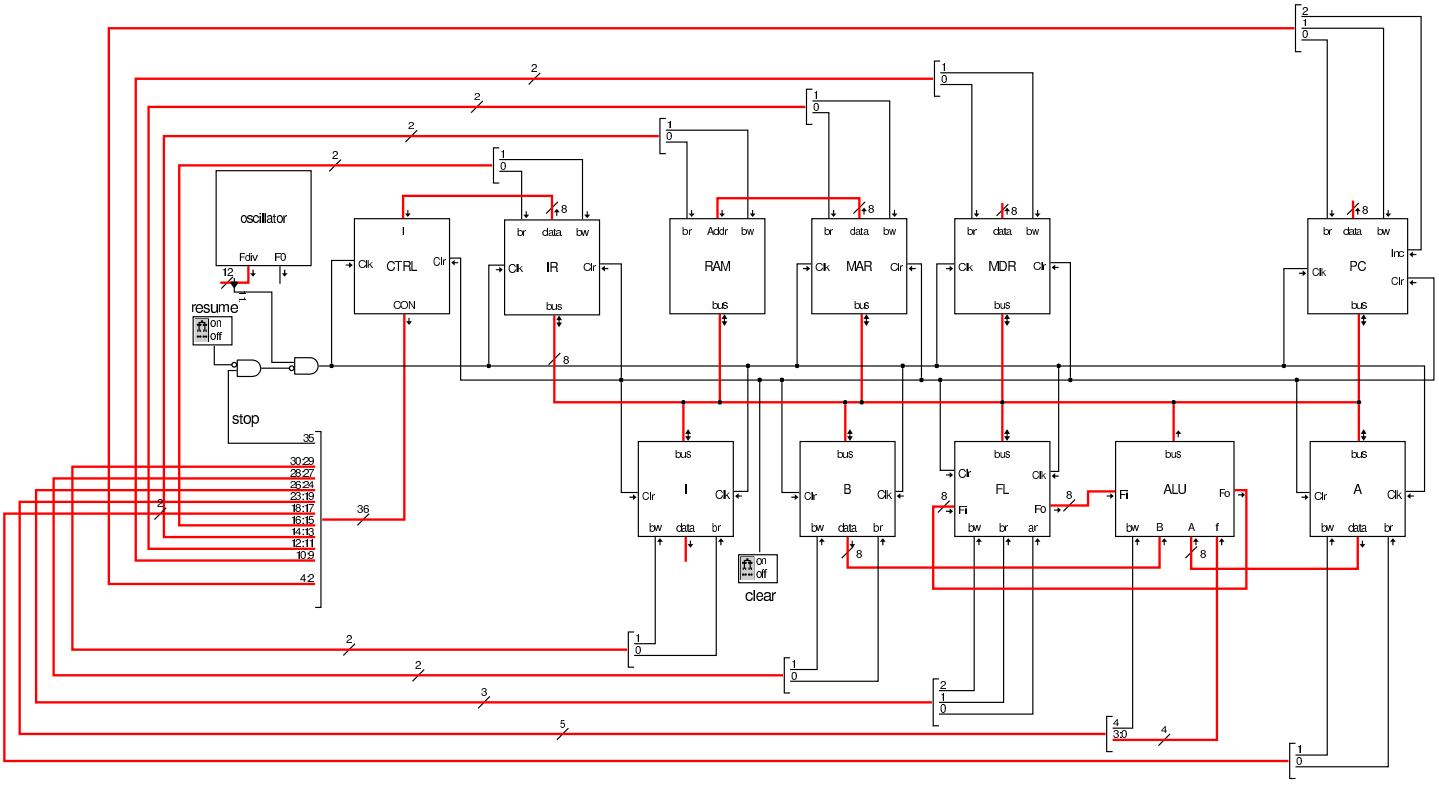
Figure 8.96. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=VRd8ilJbK_Y*



8.5 Version "A5": flags

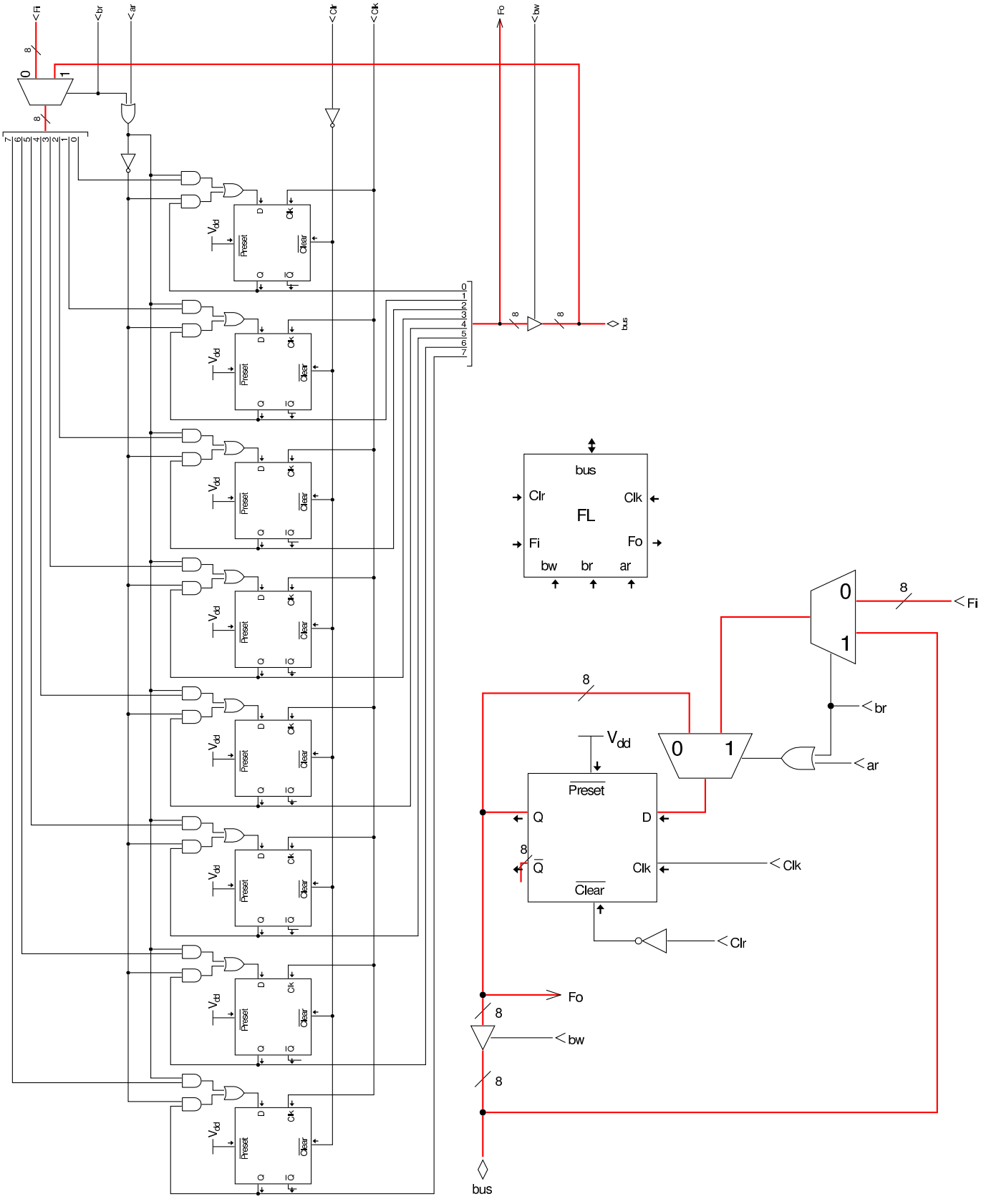
The fifth version of the simple CPU has a new special register: ***FL***, used to store the status flags. With the ***FL*** register the ALU can use previous carry (or borrow) for calculations and for rotations.

Figure 8.97. The simple CPU version “A5”.



The ***FL*** register can communicate with the bus, but it has an additional input, ***Fi***, which receives the updated status from the ALU. On the other hand, the output ***Fo*** sends to the ALU the current status of flags. The control input ***ar*** (*ALU read*) is used to let the ***FL*** register load its value from the ALU.

Figure 8.98. Internal structure of the ***FL*** register: the eight aligned modules are D flip-flops. The diagram on the right is a compact version of the same circuit.



Inside the code that describes the microcode word fields, the following fields are added, except ***fl_ar*** that already appeared in previous version. The new fields are used specifically to manage the ***FL*** register:

```
field fl_ar[24];          // FL <-- ALU
field fl_br[25];          // FL <-- bus
field fl_bw[26];          // FL --> bus
```

Inside the opcode list there are new instructions and the same is for the microcode description:

```
op move_mdr_fl {
  map move_mdr_fl : 9;            // move MDR to FL
  +0[7:0]=9;
  operands op_0;
};
op move_fl_mdr {
  map move_fl_mdr : 10;           // move FL to MDR
  +0[7:0]=10;
  operands op_0;
};
op rotcl {
  map rotcl : 42;                 // A = A rotate carry left
  +0[7:0]=42;
  operands op_0;
};
op rotcr {
  map rotcr : 43;                 // A = A rotate carry right
  +0[7:0]=43;
  operands op_0;
};
op add_carry {
  map add_carry : 44;             // A = A + B + carry
  +0[7:0]=44;
  operands op_0;
};
```

```
op sub_borrow {
  map sub_borrow : 45;              // A = A - B - borrow
  +0[7:0]=45;
  operands op_0;
};
```

```
begin microcode @ 0
…
//
move_mdr_fl:
    fl_br mdr_bw;                   // FL <-- MDR
    ctrl_start ctrl_load;           // CNT <-- 0
//
move_fl_mdr:
    mdr_br fl_bw;                   // MDR <-- FL
    ctrl_start ctrl_load;           // CNT <-- 0
//
rotcl:
    a_br alu_f=rotate_carry_left alu_bw fl_ar; // A <-- A rot. carry l
    ctrl_start ctrl_load;                     // CNT <-- 0
//
rotcr:
    a_br alu_f=rotate_carry_right alu_bw fl_ar; // A <-- A rot. carry r
    ctrl_start ctrl_load;                      // CNT <-- 0
//
add_carry:
    a_br alu_f=a_plus_b_carry alu_bw fl_ar; // A <-- A + B + carry
    ctrl_start ctrl_load;                  // CNT <-- 0
//
sub_borrow:
    a_br alu_f=a_minus_b_borrow alu_bw fl_ar; // A <-- A - B - borrow
    ctrl_start ctrl_load;                    // CNT <-- 0
…
end
```

Figure 8.102. Match between the contents of the memory that represents the microcode (the couple of memory banks **m1** and **m2** inside the control unit) and the microcode description itself.

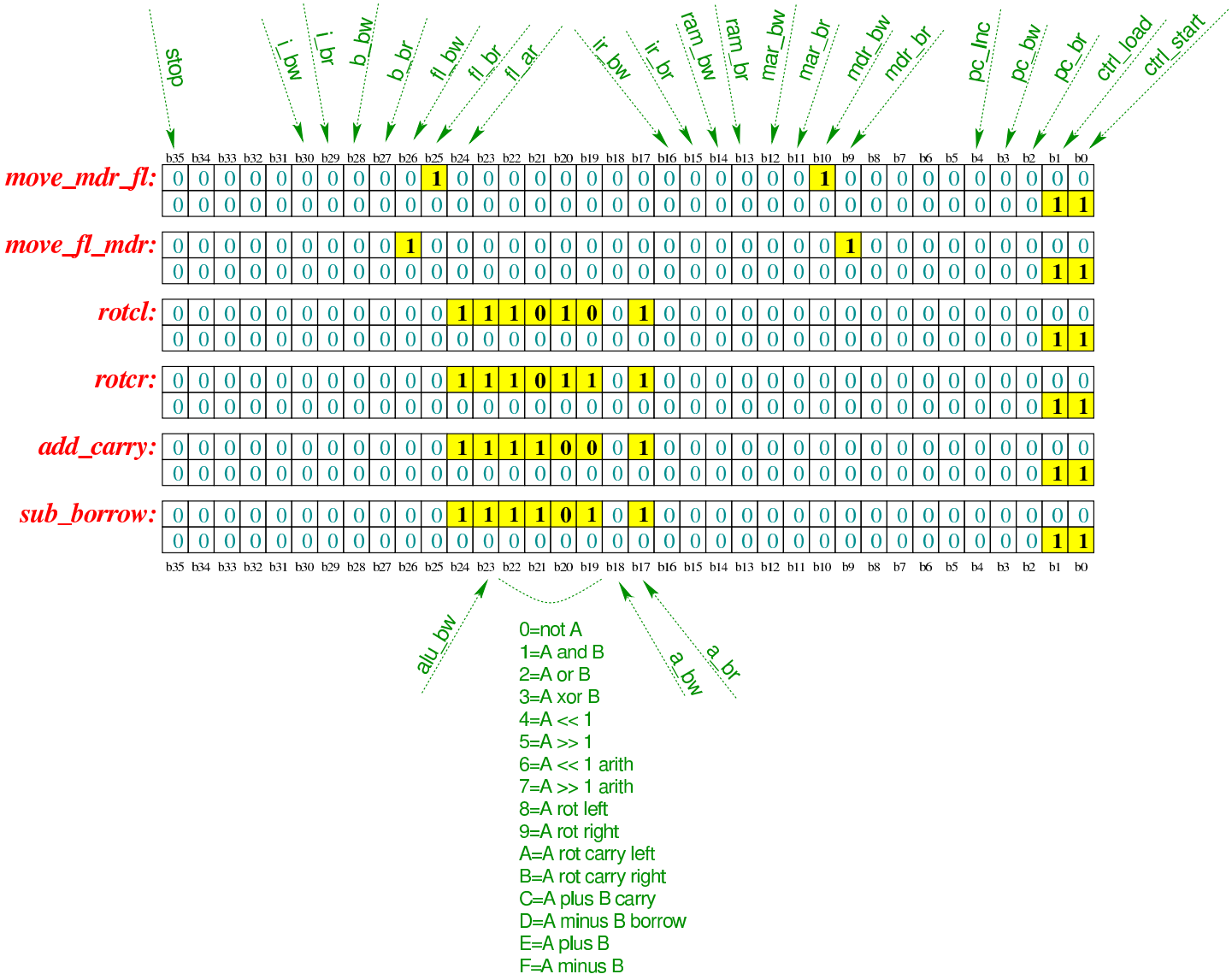


Table 8.103. Macroinstructions added for this version of the simple CPU.

Syntax	Description
`move_mdr_fl`	It copies the ***MDR*** content into the ***FL*** register.
`move_fl_mdr`	It copies the ***FL*** content into the ***MDR*** register.

Syntax	Description
`rotcl`	It does a binary left rotation of the ***A*** register, using also the carry flag: the previous carry shifts in on the right and the most significant bit is shifted out to the new carry value.
`rotcr`	It does a binary right rotation of the ***A*** register, using also the carry flag: the previous carry shifts in on the left and the least significant bit is shifted out to the new carry value.
`add_carry`	It adds ***A*** and ***B*** and the previous carry, updating the ***A*** register and the new carry status.
`sub_borrow`	It subtracts ***A*** and ***B*** and the previous borrow, updating the ***A*** register and the new borrow status (carry and borrow are the same flag).

The following sections offer various examples, where the use of all the new instructions is shown.

8.5.1 Instruction "rotcl" and "rotcr"

«

Listing 8.104. Macrocode example to experiment the rotation with carry: a value is loaded from the memory to the ***A*** register and a left rotation with carry is done five times, updating the value of ***A***. Then the result is copied into ***B***, before another value is loaded and rotated five times right. The complete source file should be available at *attachments/xcpu/xcpu-a5-rotc.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_1
        move_mdr_a
        rotcl
        rotcl
        rotcl
        rotcl
        rotcl
        move_a_mdr
        move_mdr_b
```

```
        rotcr
        rotcr
        rotcr
        rotcr
        rotcr
stop:
        stop
data_1:
        .byte 160
end
```

Figure 8.105. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value.

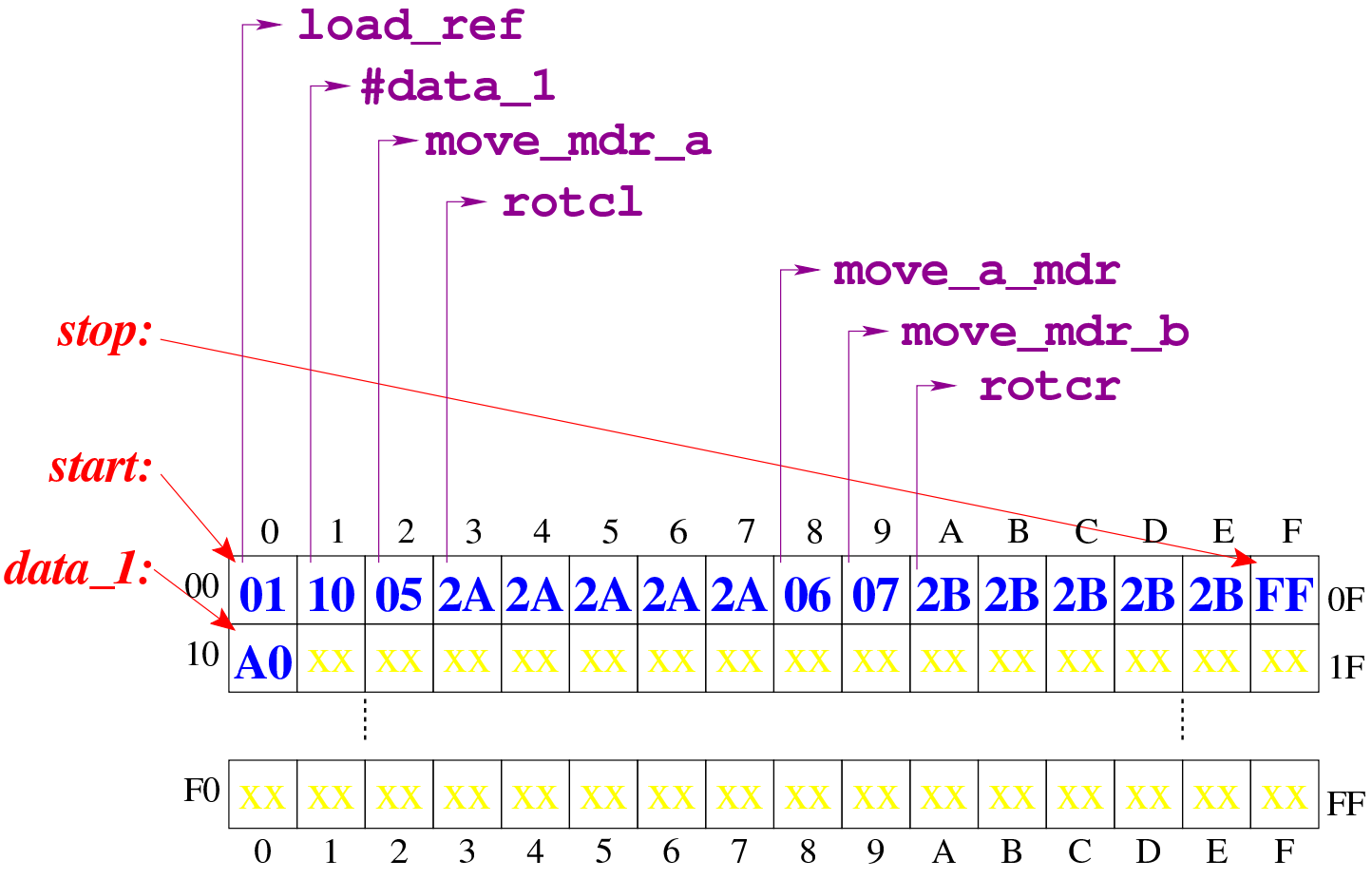
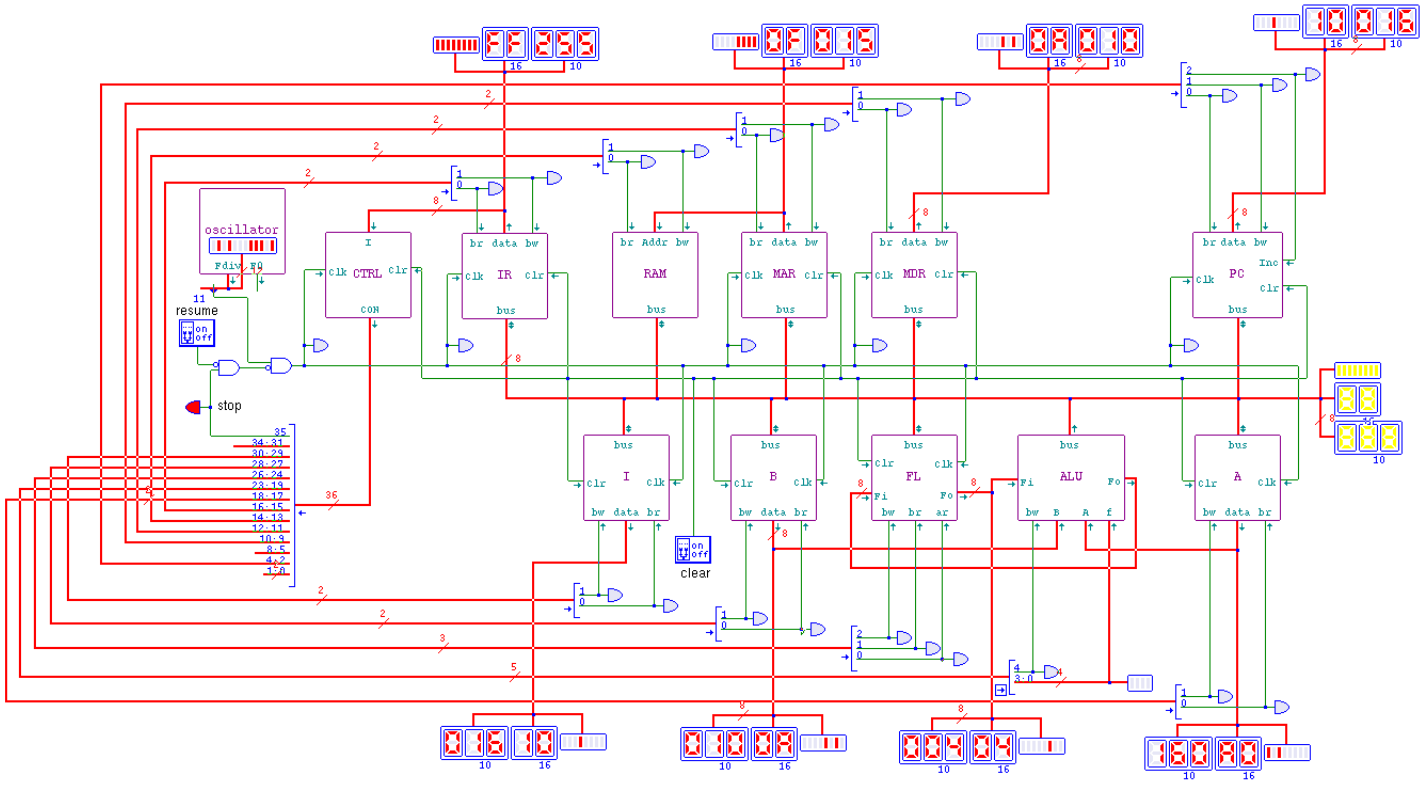


Figure 8.106. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=Zl3d-Tg5C1Q*



8.5.2 Instruction "add_carry"

Listing 8.107. Macrocode example to experiment the addition with carry: two 16-bit numbers should be added, splitting them and adding in two steps. The numbers are $12FF_{16}$ and $11EE_{16}$. First there is the sum of FF_{16} and EE_{16}, without considering the previous carry, then there is the sum of 12_{16} and 11_{16} with the previous carry. The result is ED_{16} plus a carry the first time, where the second time the result is 24_{16}. Every partial addition is saved in memory, so that at the address matching the label **`data_4`** a 16-bit result, little-endian, can be read. The complete source file should be available at *attachments/xcpu/xcpu-a5-addc.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_0
        move_mdr_a
        load_ref #data_2
        move_mdr_b
        add
```

```
        move_a_mdr
        store_ref #data_4
        load_ref #data_1
        move_mdr_a
        load_ref #data_3
        move_mdr_b
        add_carry
        move_a_mdr
        store_ref #data_5
stop:
        stop
// 0x12FF = 4863
data_0:
        .byte 0xFF
data_1:
        .byte 0x12
// 0x11EE = 4590
data_2:
        .byte 0xEE
data_3:
        .byte 0x11
data_4:
        .byte 0
data_5:
        .byte 0
end
```

Figure 8.108. RAM memory content before the program execution: the memory cells marked with "xx" have an unknown and unspecified value.

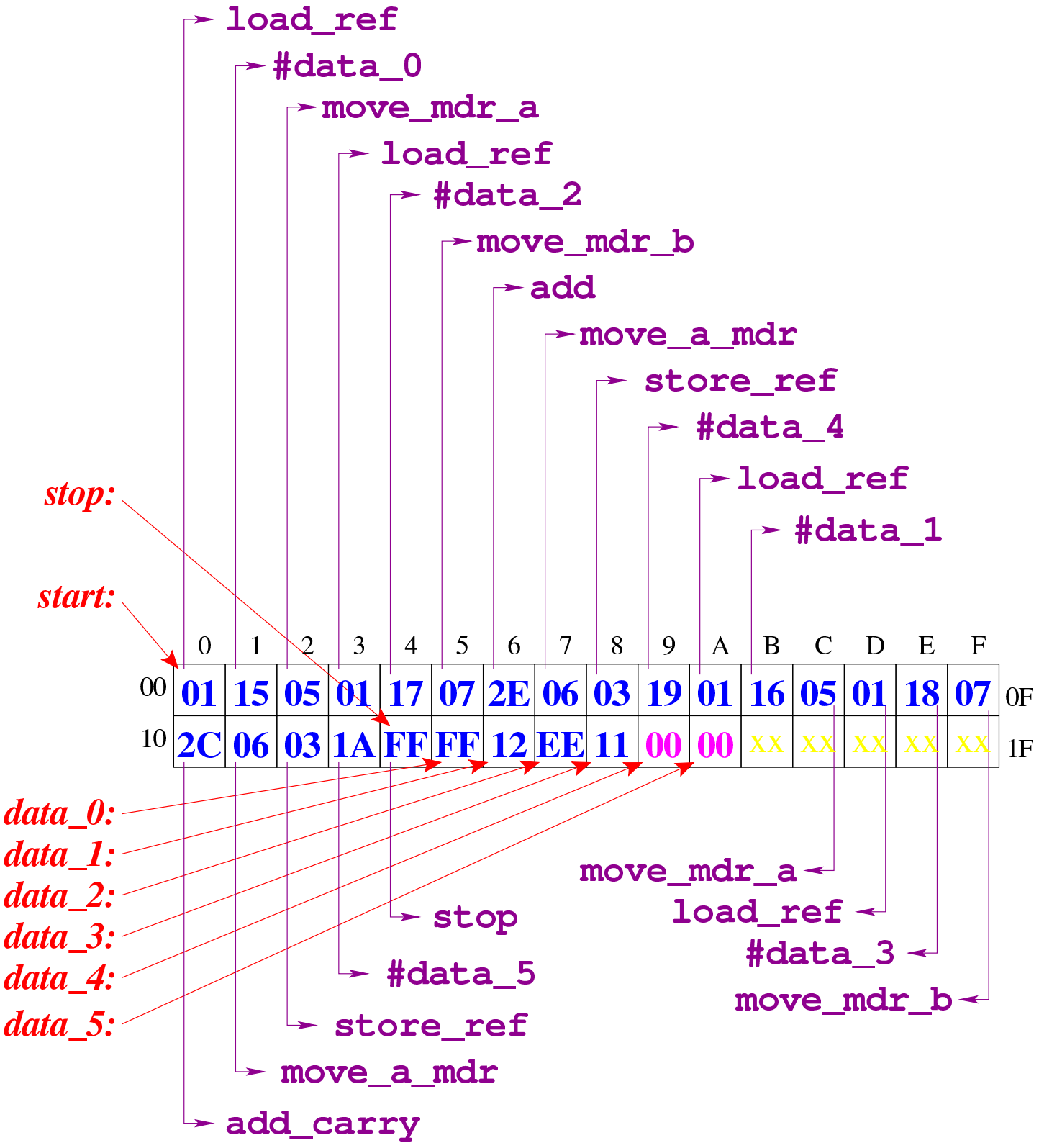


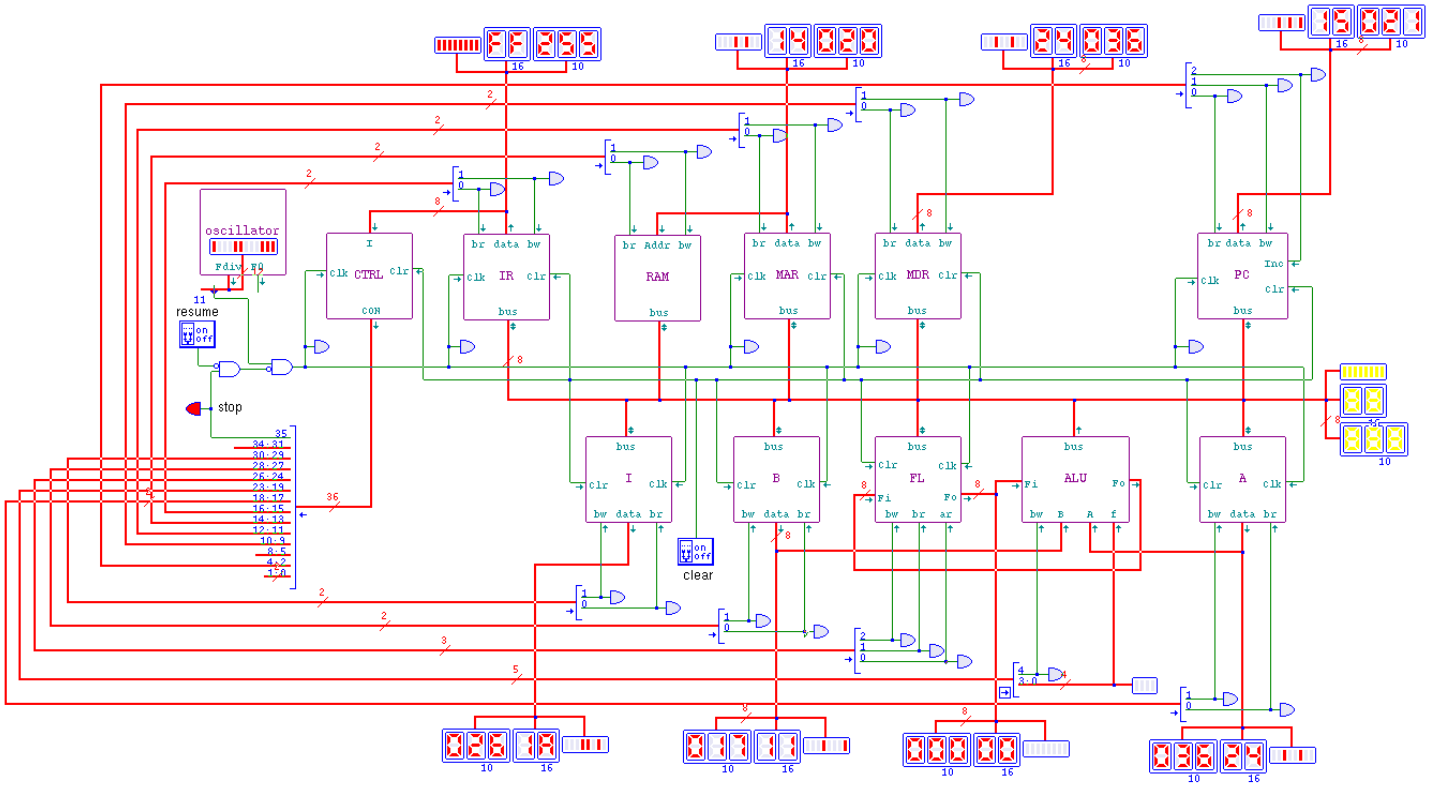
Figure 8.109. At the end of the macroprogram, the result is stored in memory.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	01	15	05	01	17	07	2E	06	03	19	01	16	05	01	18	07	0F
10	2C	06	03	1A	FF	FF	12	EE	11	ED	24	XX	XX	XX	XX	XX	1F

data_4:

data_5:

Figure 8.110. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=1Xu4MxWBwW4*



8.5.3 Instruction "sub_borrow"

«

Listing 8.111. Macrocode example to experiment the subtraction with borrow: the subtraction $12EE_{16}-11FF_{16}$ is to be calculated. First a normal subtraction $EE_{16}-FF_{16}$ is done, saving the request of borrow for the next subtraction: $12_{16}-11_{16}-\textbf{\textit{borrow}}$. The first subtraction gives EF_{16} with a borrow request; the second subtraction gives 00_{16}. Every partial subtraction is saved in memory, so that at the address matching the label **data_4** a 16-bit result, little-endian, can be read. The complete source file should be available at *attachments/xcpu/xcpu-a5-subb.gm* .

```
begin macrocode @ 0
start:
        load_ref #data_0
        move_mdr_a
        load_ref #data_2
        move_mdr_b
        sub
        move_a_mdr
        store_ref #data_4
        load_ref #data_1
        move_mdr_a
        load_ref #data_3
        move_mdr_b
        sub_borrow
        move_a_mdr
        store_ref #data_5
stop:
        stop
// 0x12EE = 4846
data_0:
        .byte 0xEE
data_1:
        .byte 0x12
// 0x11FF = 4607
data_2:
```

```
        .byte 0xFF
data_3:
        .byte 0x11
data_4:
        .byte 0
data_5:
        .byte 0
end
```

Figure 8.112. RAM memory content before the program execution: the memory cells marked with "xx" have an unknown and unspecified value.

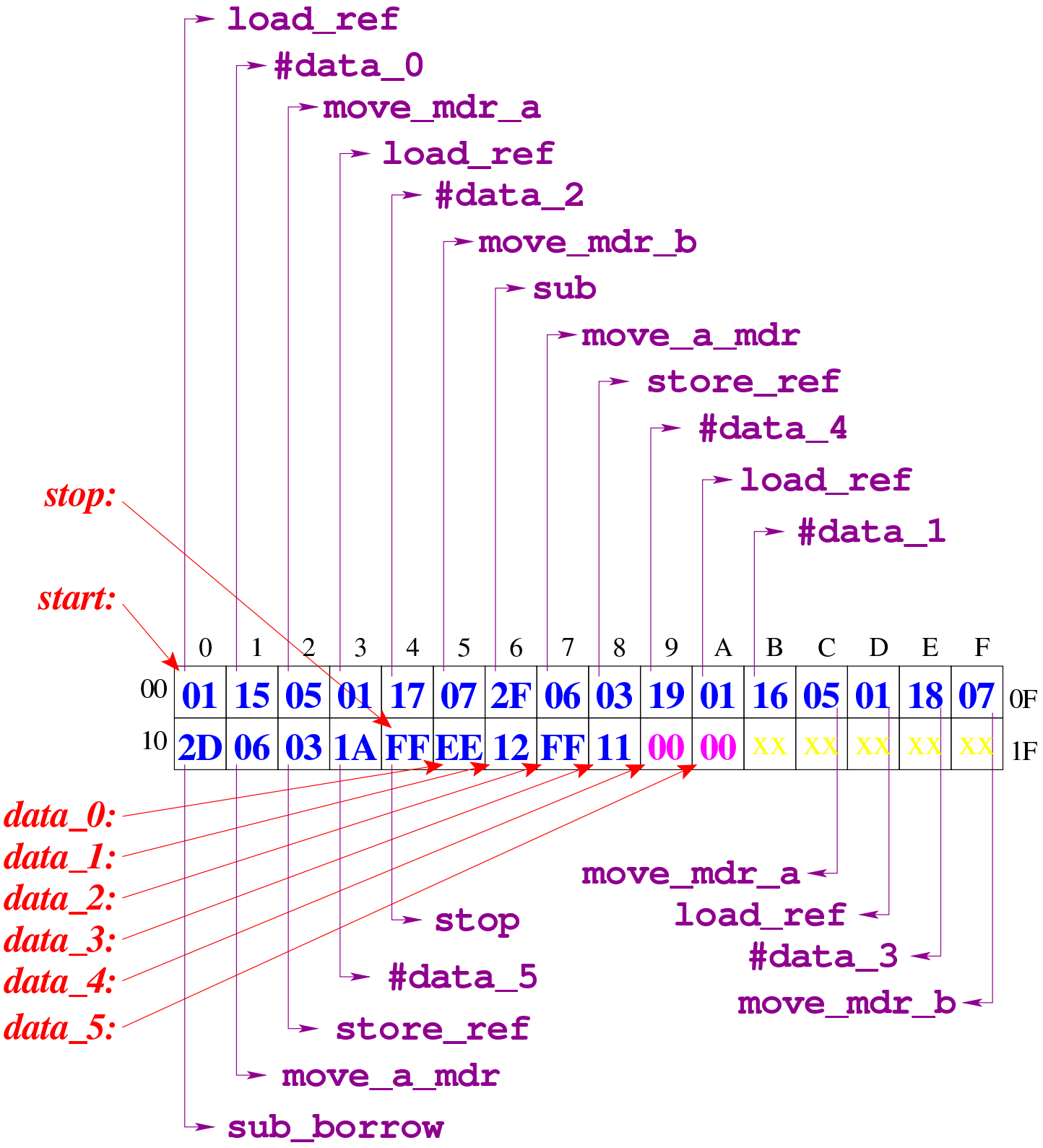


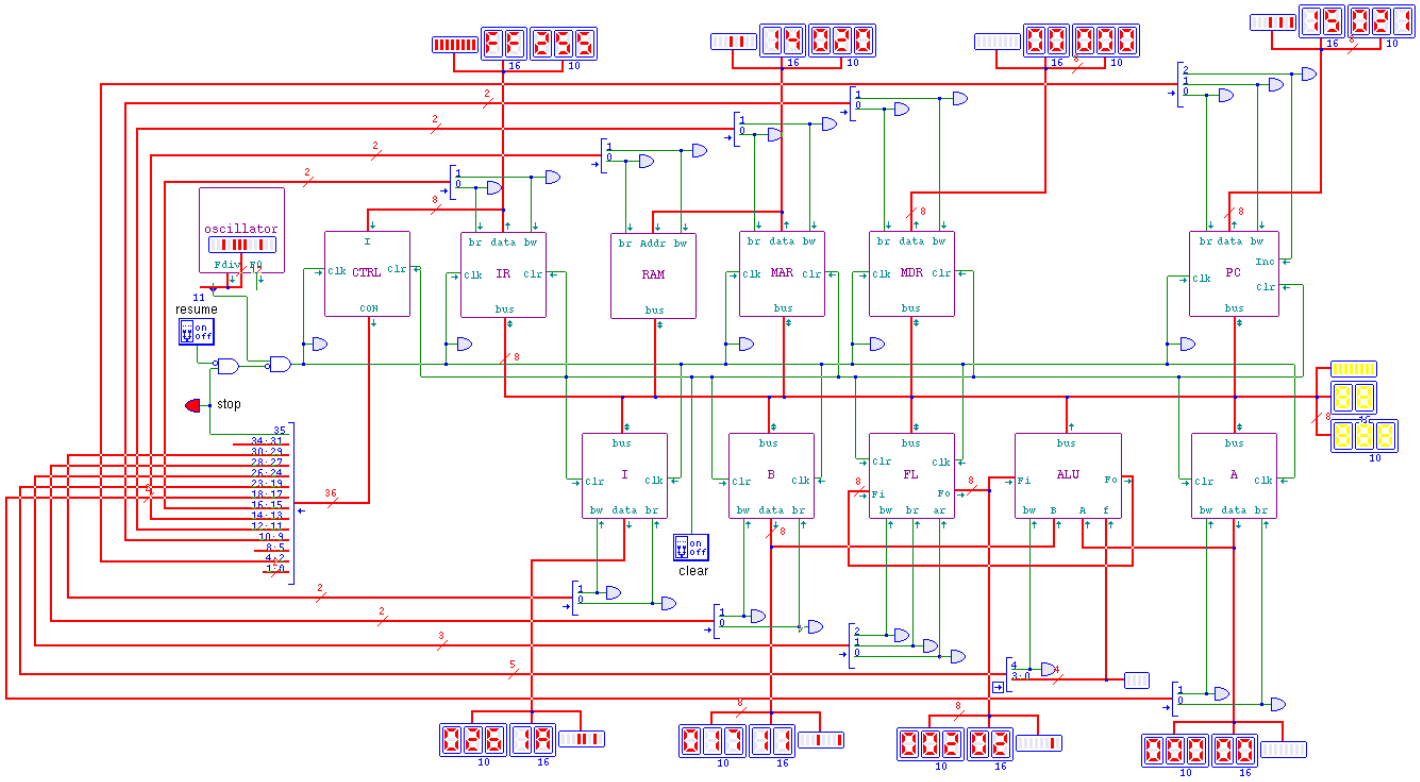
Figure 8.113. At the end of the macroprogram, the result is stored in memory.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	01	15	05	01	17	07	2F	06	03	19	01	16	05	01	18	07	0F
10	2D	06	03	1A	FF	FF	12	EE	11	EF	00	XX	XX	XX	XX	XX	1F

data_4:

data_5:

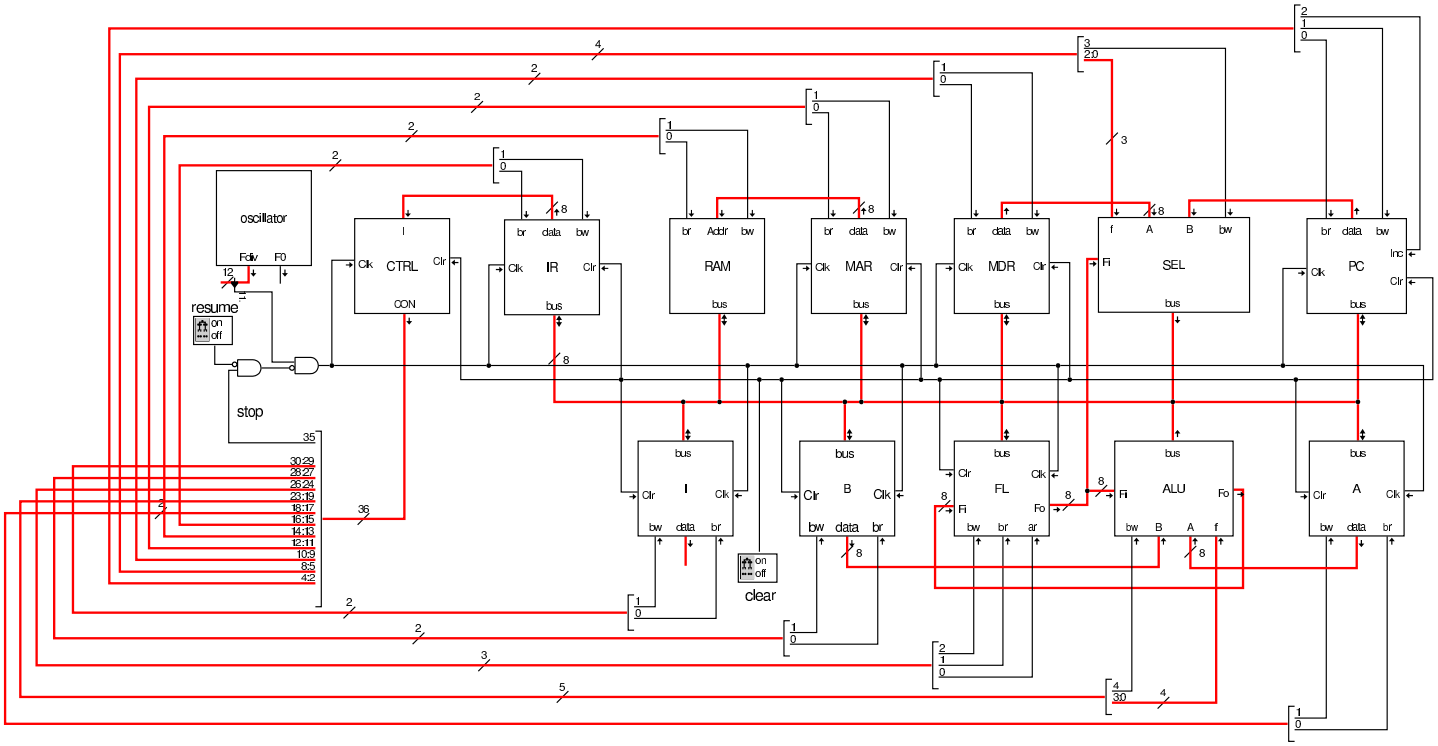
Figure 8.114. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=ofPUzdIids8*



8.6 Version "A6": branching

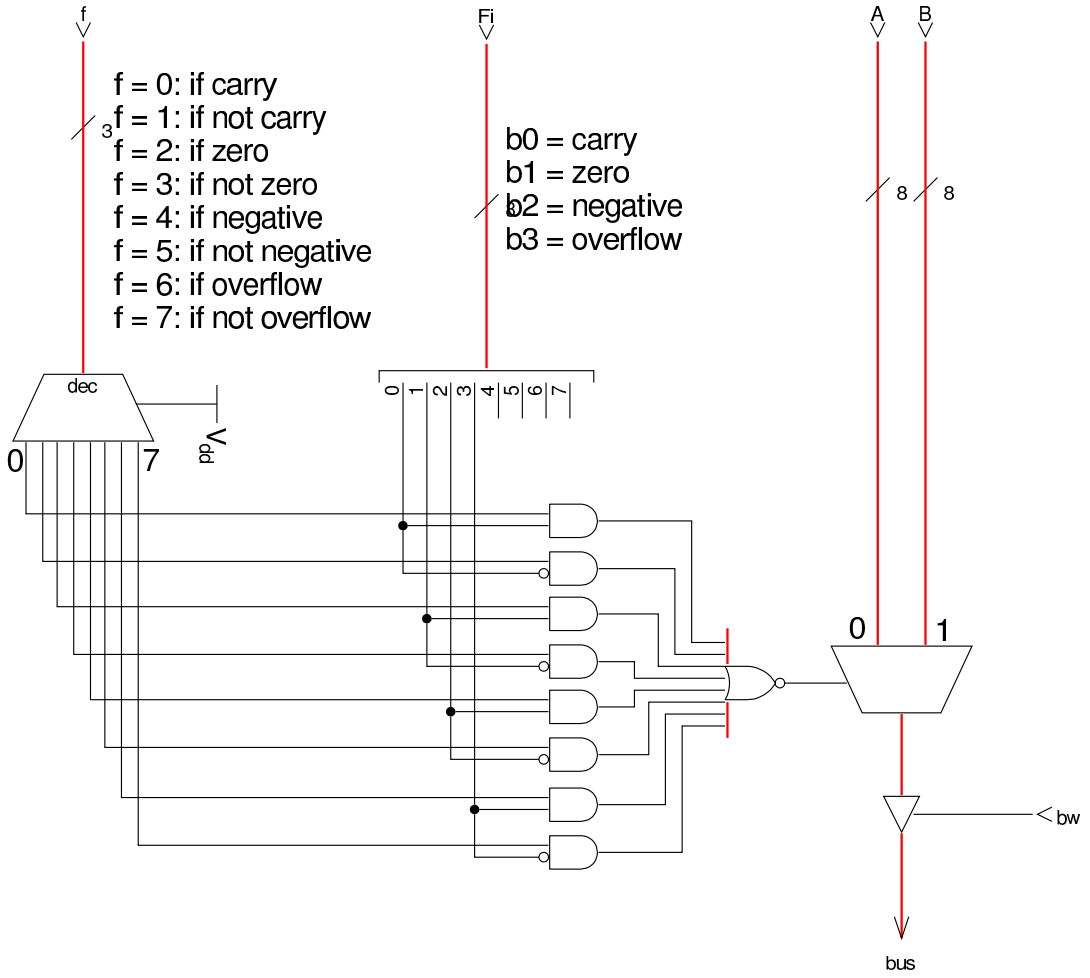
In the sixth version of the simple CPU a new module is added, with the purpose to test conditions on the value of the status flags (the ***FL*** register). The condition evaluation results in the choice from the value of two registers: ***MDR*** if the condition is true, or ***PC*** otherwise.

Figure 8.115. Simple CPU version “A6”.



The **SEL** module gets two values from input ***A*** and input ***B***. From the input ***Fi*** it gets the flags status, as it is output from the ***FL*** register. From the input ***f*** the module **SEL** receives the condition to be checked: if the condition is true and the input ***bw*** is asserted, the module writes to the bus the same value obtained from the ***A*** input, otherwise, if the condition results false, the module writes to the bus the same value obtained from the ***B*** input.

Figure 8.116. Module **SEL**, internal structure.



Inside the code that describes the microcode word fields, the following fields are added. The new fields are used specifically to manage the **SEL** module:

```
field sel_f[7:5]={
        if_carry=0,
        if_not_carry=1,
        if_zero=2,
        if_not_zero=3,
        if_negative=4,
        if_not_negative=5,
        if_overflow=6,
        if_not_overflow=7
};
field sel_bw[8];          // SEL --> bus
```

Inside the opcode list there are new instructions and the same is for the microcode description:

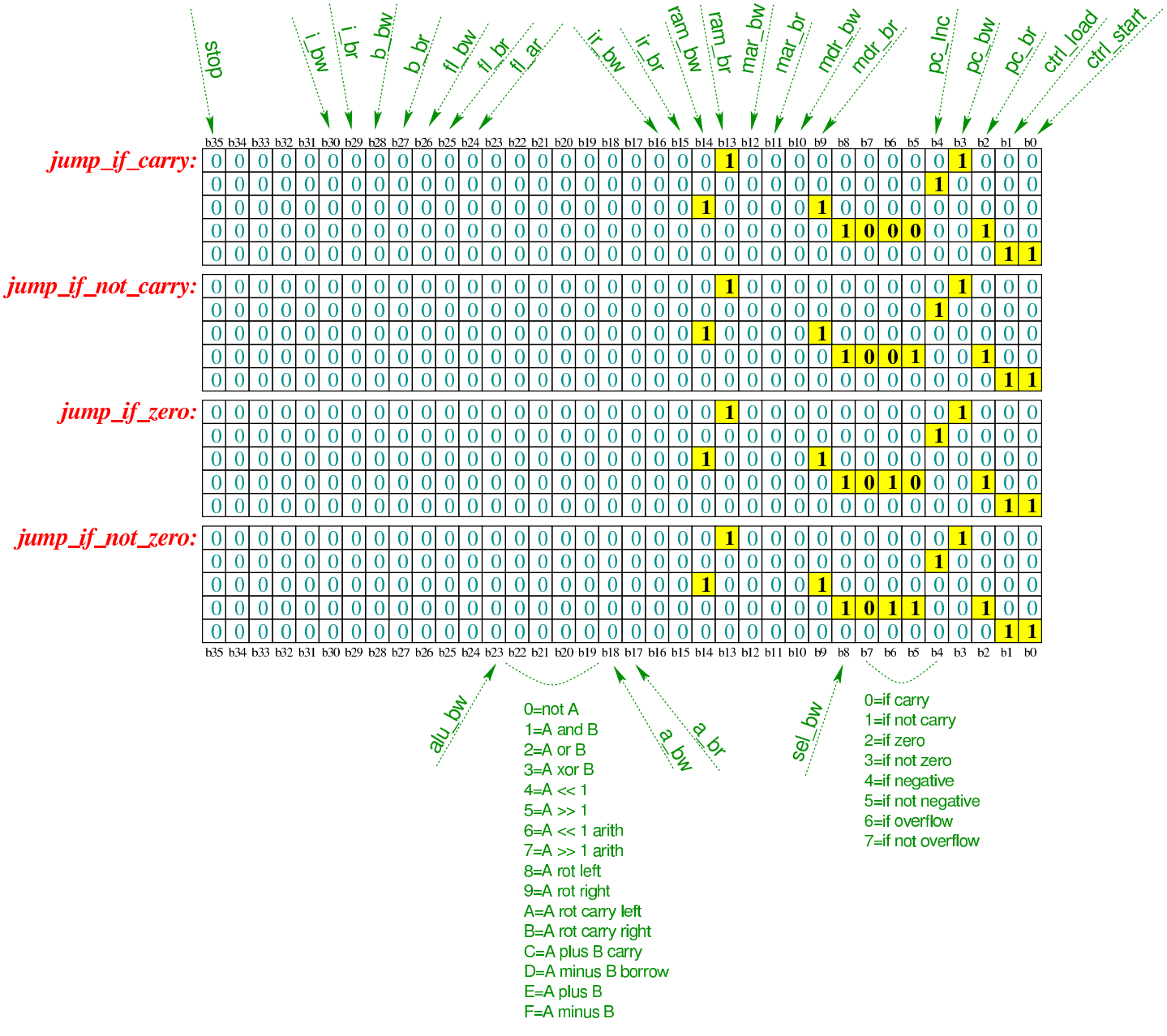
```
op jump_if_carry {
  map jump_if_carry : 16;          // jump to #nn if carry==1
  +0[7:0]=16;
  operands op_1;
};
op jump_if_not_carry {
  map jump_if_not_carry : 17;      // jump to #nn if carry==0
  +0[7:0]=17;
  operands op_1;
};
op jump_if_zero {
  map jump_if_zero : 18;           // jump to #nn if zero==1
  +0[7:0]=18;
  operands op_1;
};
op jump_if_not_zero {
  map jump_if_not_zero : 19;       // jump to #nn if zero==0
  +0[7:0]=19;
  operands op_1;
};
op jump_if_negative {
  map jump_if_negative : 20;       // jump to #nn if negative==1
  +0[7:0]=20;
  operands op_1;
};
op jump_if_not_negative {
  map jump_if_not_negative : 21; // jump to #nn if negative==0
  +0[7:0]=21;
  operands op_1;
};
op jump_if_overflow {
  map jump_if_overflow : 22;       // jump to #nn if overflow==1
  +0[7:0]=22;
  operands op_1;
```

```
};
op jump_if_not_overflow {
  map jump_if_not_overflow : 23; // jump to #nn if overflow==0
  +0[7:0]=23;
  operands op_1;
};
```

```
begin microcode @ 0
…
jump_if_carry:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_carry sel_bw         // PC = (carry ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_not_carry:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_not_carry sel_bw     // PC = (not_carry ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_zero:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_zero sel_bw          // PC = (zero ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_not_zero:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_not_carry sel_bw     // PC = (not_zero ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_negative:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
```

```
    mdr_br ram_bw;                          // MDR <-- RAM[mar]
    pc_br sel_f=if_negative sel_bw          // PC = (negative ? MAR : PC)
    ctrl_start ctrl_load;                   // CNT <-- 0
//
jump_if_not_negative:
    mar_br pc_bw;                           // MAR <-- PC
    pc_Inc;                                 // PC++
    mdr_br ram_bw;                          // MDR <-- RAM[mar]
    pc_br sel_f=if_not_negative sel_bw      // PC = (not_negative ? MAR : PC)
    ctrl_start ctrl_load;                   // CNT <-- 0
//
jump_if_overflow:
    mar_br pc_bw;                           // MAR <-- PC
    pc_Inc;                                 // PC++
    mdr_br ram_bw;                          // MDR <-- RAM[mar]
    pc_br sel_f=if_overflow sel_bw          // PC = (overflow ? MAR : PC)
    ctrl_start ctrl_load;                   // CNT <-- 0
//
jump_if_not_overflow:
    mar_br pc_bw;                           // MAR <-- PC
    pc_Inc;                                 // PC++
    mdr_br ram_bw;                          // MDR <-- RAM[mar]
    pc_br sel_f=if_not_overflow sel_bw      // PC = (not_overflow ? MAR : PC)
    ctrl_start ctrl_load;                   // CNT <-- 0
…
end
```

Figure 8.120. Match between the contents of the memory that represents the microcode (the couple of memory banks **`m1`** and **`m2`** inside the control unit) and the microcode description itself.



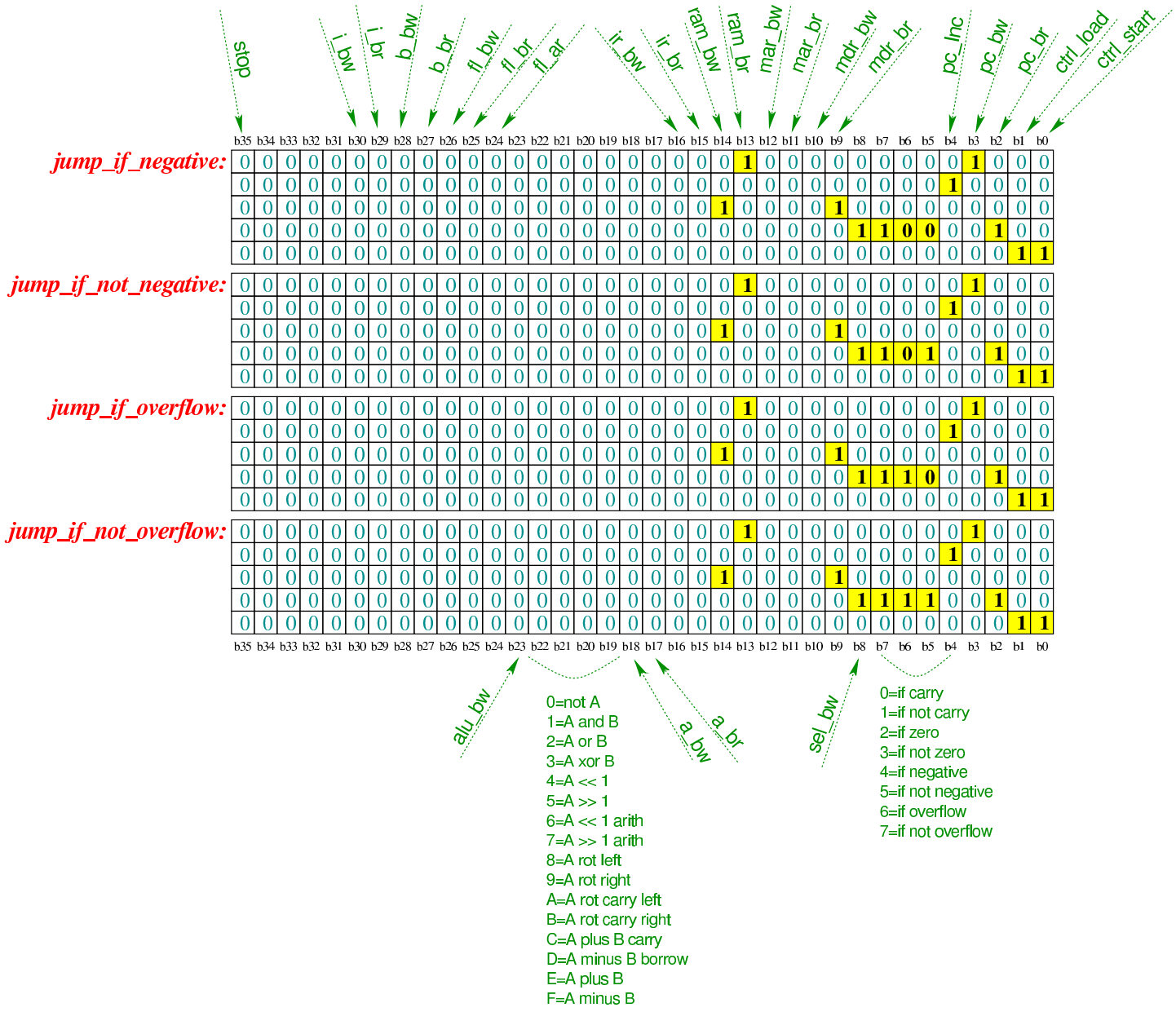


Table 8.121. Macroinstructions added for this version of the simple CPU.

Syntax	Description
`jump_if_carry #`***ref***	If the carry flag is true, it jumps to the specified address.
`jump_if_not_carry #`***ref***	If the carry flag is false, it jumps to the specified address.
`jump_if_zero #`***ref***	If the zero flag is true, it jumps to the specified address.
`jump_if_not_zero #`***ref***	If the zero flag is false, it jumps to the specified address.

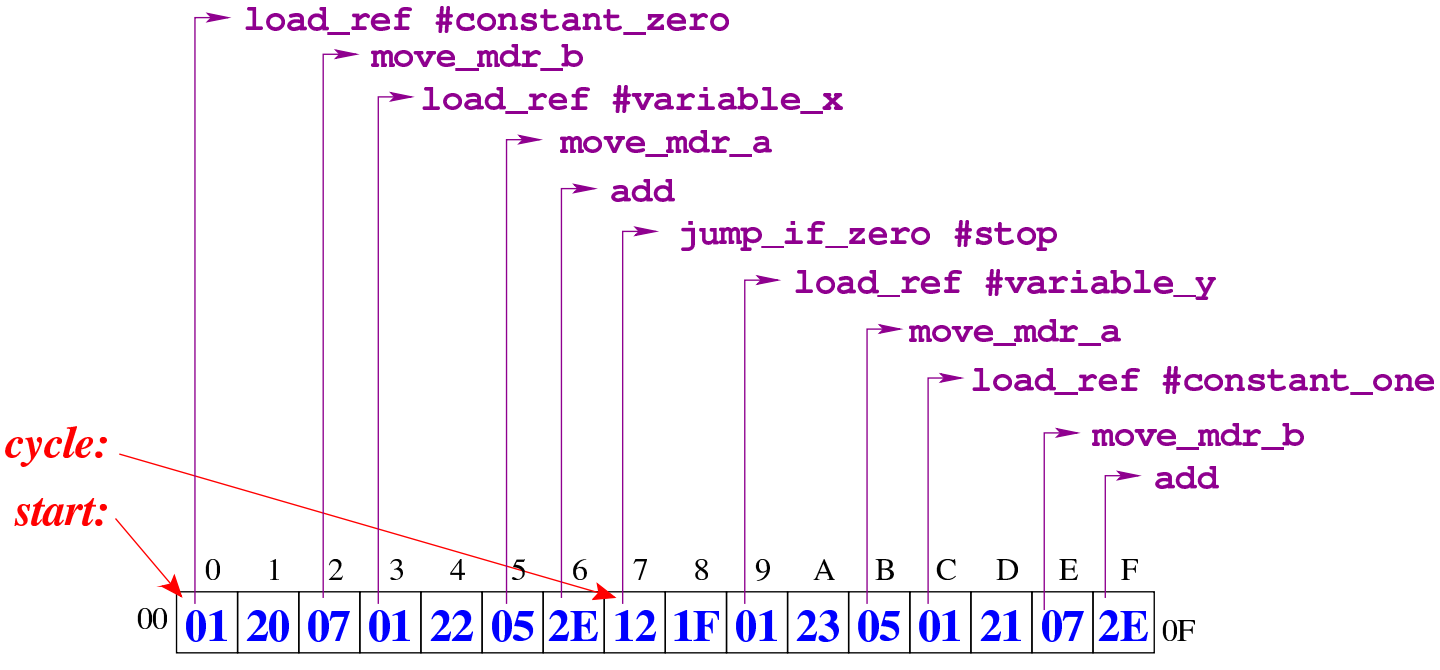
Syntax	Description
`jump_if_negative #`***ref***	If the negative flag is true, it jumps to the specified address.
`jump_if_not_negative #`***ref***	If the negative flag is false, it jumps to the specified address.
`jump_if_overflow #`***ref***	If the overflow flag is true, it jumps to the specified address.
`jump_if_not_overflow #`***ref***	If the overflow flag is false, it jumps to the specified address.

Listing 8.122. Macrocode example to experiment the selection module: there is a cycle there a variable is incremented until it contains the result of a sum with another variable. The complete source file should be available at *attachments/xcpu/xcpu-a6.gm* .

```
begin macrocode @ 0
start:
        load_ref #constant_zero
        move_mdr_b
        load_ref #variable_x
        move_mdr_a
        add
cycle:
        jump_if_zero #stop
        load_ref #variable_y
        move_mdr_a
        load_ref #constant_one
        move_mdr_b
        add
        move_a_mdr
        store_ref #variable_y
        load_ref #variable_x
        move_mdr_a
        load_ref #constant_one
        move_mdr_b
        sub
```

```
        move_a_mdr
        store_ref #variable_x
        jump #cycle
stop:
        stop
constant_zero:
        .byte 0
constant_one:
        .byte 1
variable_x:
        .byte 3
variable_y:
        .byte 7
end
```

Figure 8.123. RAM memory content before the program execution: the memory cells marked with "xx" have an unknown and unspecified value. At the end of the execution, the memory cell at the address 23_{16} represents ***variable_y*** and it contains the value $0A_{16}$ (10_{10}).



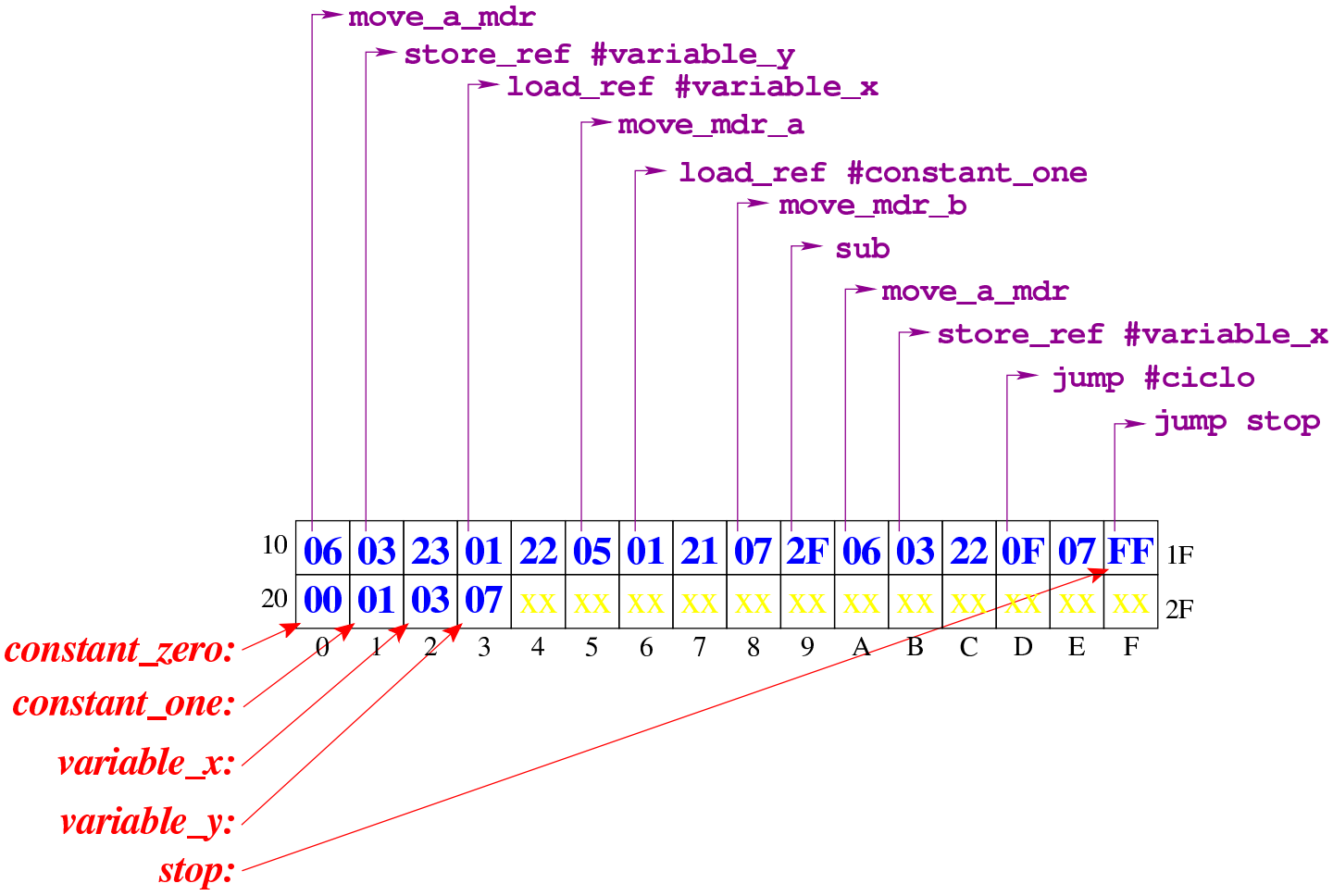
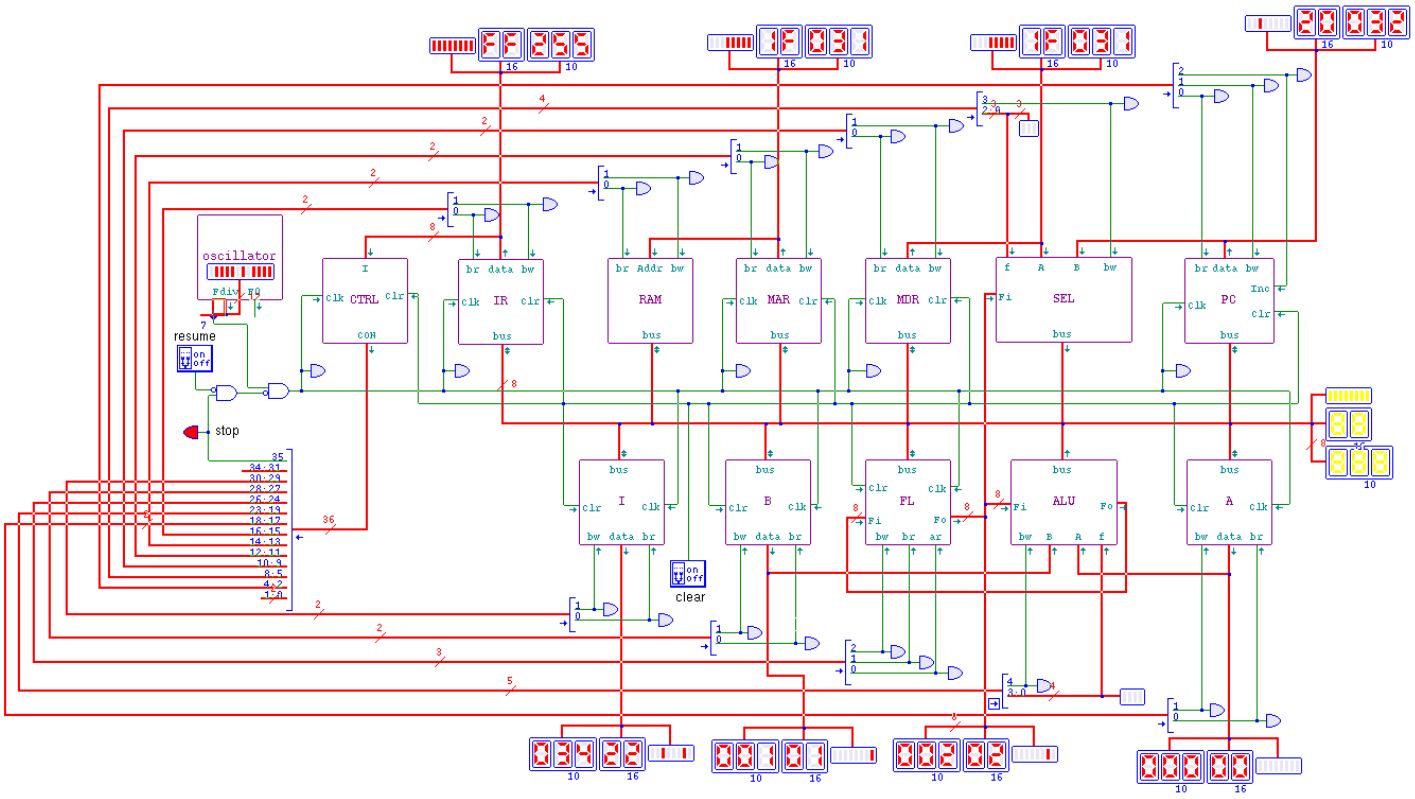

move_a_mdr
store_ref #variable_y
load_ref #variable_x
move_mdr_a
load_ref #constant_one
move_mdr_b
sub
move_a_mdr
store_ref #variable_x
jump #ciclo
jump stop
10 06 03 23 01 22 05 01 21 07 2F 06 03 22 0F 07 FF 1F
20 00 01 03 07 XX XX XX XX XX XX XX XX XX XX XX XX 2F
0 1 2 3 4 5 6 7 8 9 A B C D E F
constant_zero:
constant_one:
variable_x:
variable_y:
stop:

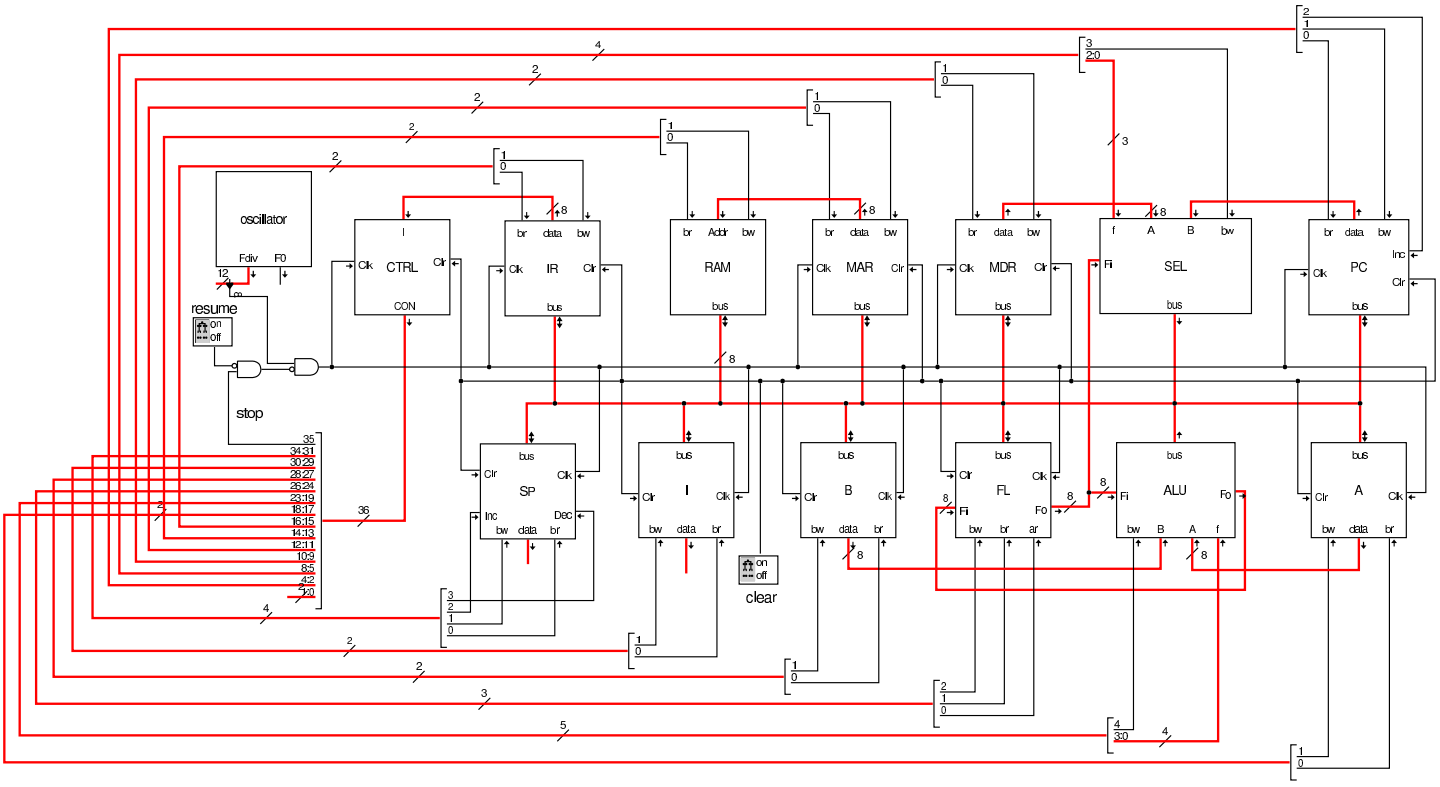
Figure 8.124. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=hFoOoGf86t0*



8.7 Version "A7": stack

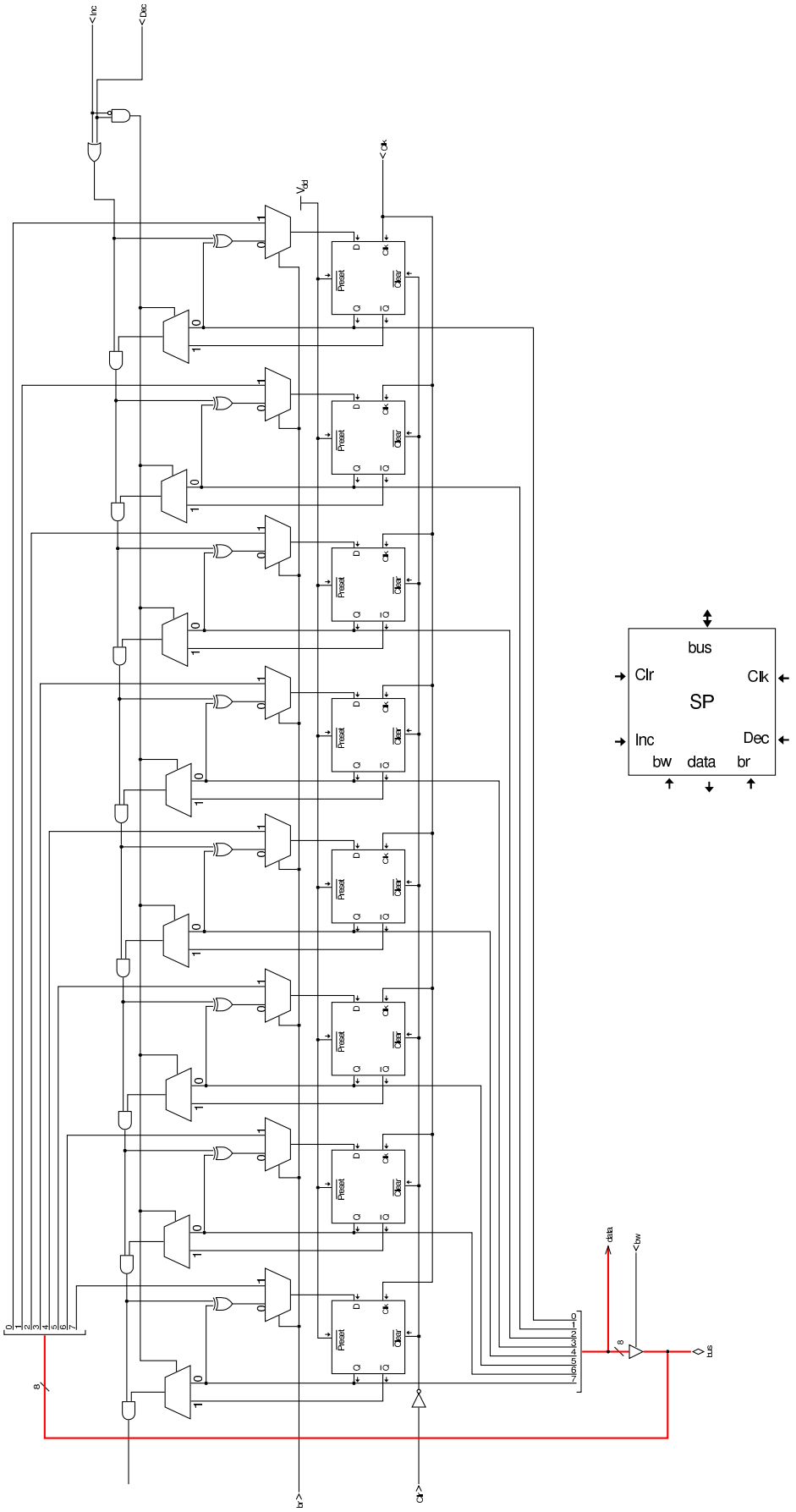
In the seventh version of the simple CPU the ***SP*** register is added (*stack pointer*). This new register is to be used as the pointer for the stack. The stack is used mainly to allow the call of subroutines, with instructions like **`call`** and **`return`**, and to save and restore the registers.

Figure 8.125. Simple CPU version “A7”.



The ***SP*** register has two extra control inputs: ***Inc*** and ***Dec***, which requires, respectively, the increment or decrement of the register content.

Figure 8.126. Register **SP**.



Inside the code that describes the microcode word fields, the following fields are added. The new fields are used specifically to manage the ***SP*** register:

```
field sp_br[31];          // SP <-- bus
field sp_bw[32];          // SP --> bus
field sp_Inc[33];         // SP++
field sp_Dec[34];         // SP--
```

Inside the opcode list there are new instructions and the same is for the microcode description:

```
op call_ref {
  map call_ref : 24;                    // call #nn
  +0[7:0]=24;
  operands op_1;
};
op call_reg {
  map call_reg : 25;                    // call I
  +0[7:0]=25;
  operands op_0;
};
op return {
  map return : 26;                      // return
  +0[7:0]=26;
  operands op_0;
};
op push_mdr {
  map push_mdr : 27;                    // push MDR
  +0[7:0]=27;
  operands op_0;
};
op push_a {
  map push_a : 28;                      // push A
  +0[7:0]=28;
  operands op_0;
};
op push_b {
```

```
  map push_b : 29;                    // push B
  +0[7:0]=29;
  operands op_0;
};
op push_fl {
  map push_fl : 30;                   // push FL
  +0[7:0]=30;
  operands op_0;
};
op push_i {
  map push_i : 31;                    // push I
  +0[7:0]=31;
  operands op_0;
};
op pop_mdr {
  map pop_mdr : 48;                   // pop MDR
  +0[7:0]=48;
  operands op_0;
};
op pop_a {
  map pop_a : 49;                     // pop A
  +0[7:0]=49;
  operands op_0;
};
op pop_b {
  map pop_b : 50;                     // pop B
  +0[7:0]=50;
  operands op_0;
};
op pop_fl {
  map pop_fl : 51;                    // pop FL
  +0[7:0]=51;
  operands op_0;
};
op pop_i {
```

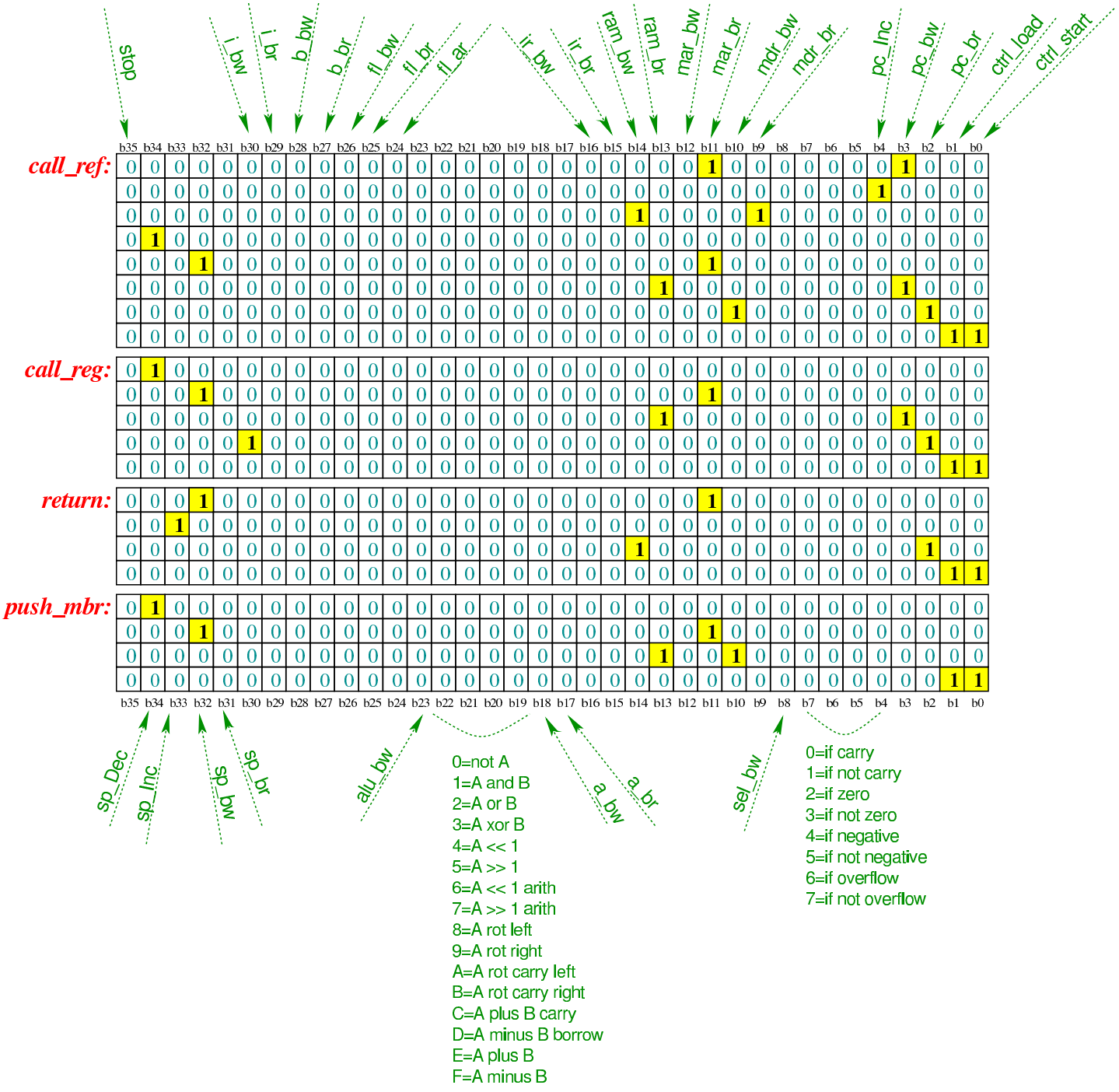
```
  map pop_i : 52;                          // pop I
  +0[7:0]=52;
  operands op_0;
};
```

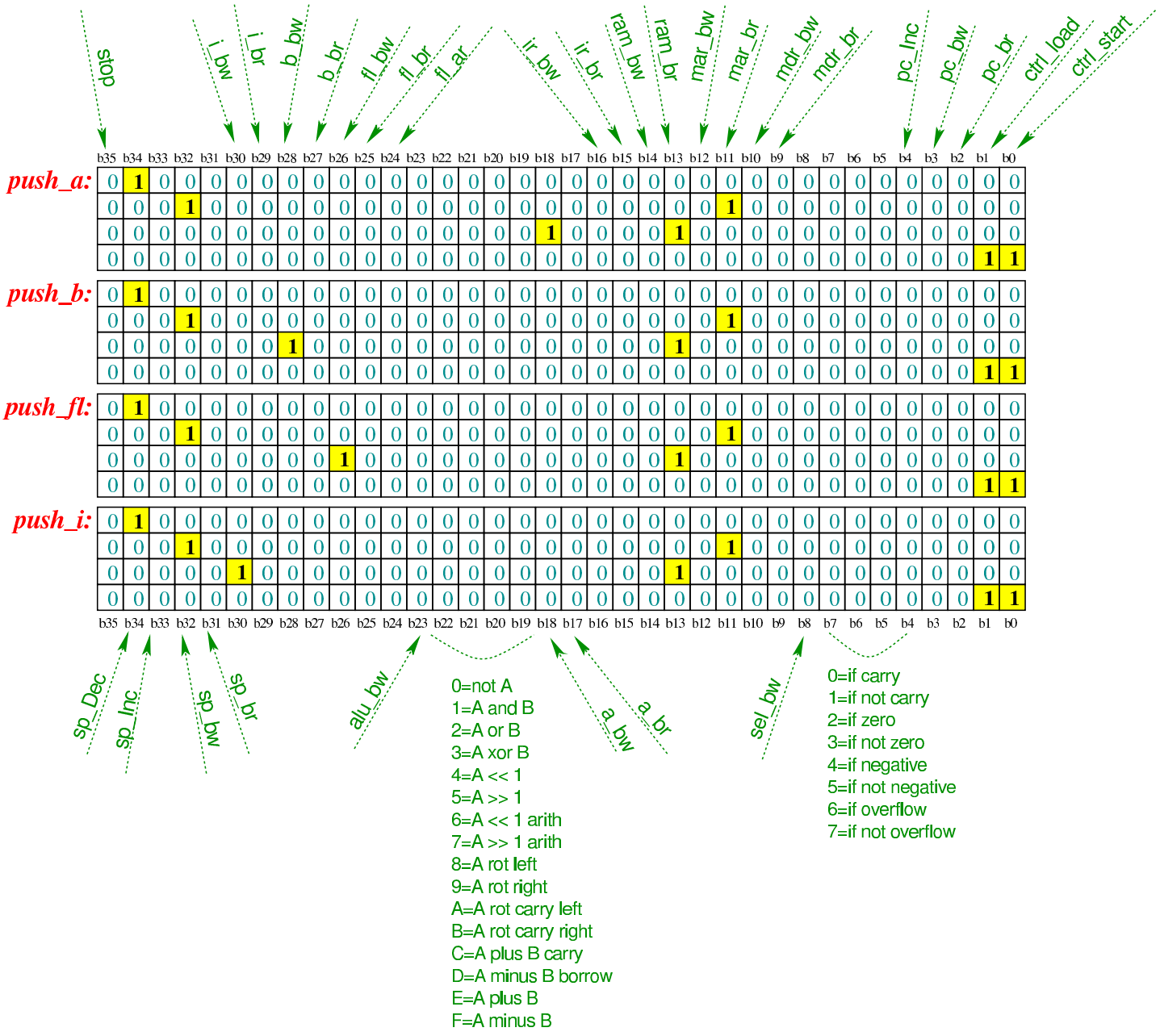
```
begin microcode @ 0
…
call_ref:
    mar_br pc_bw;                          // MAR <-- PC
    pc_Inc;                                // PC++
    mdr_br ram_bw;                         // MDR <-- RAM[mar]
    sp_Dec;                                // SC--
    mar_br sp_bw;                          // MAR <-- SP
    ram_br pc_bw;                          // RAM[mar] <-- PC
    pc_br mdr_bw;                          // PC <-- MDR
    ctrl_start ctrl_load;                  // CNT <-- 0
//
call_reg:
    sp_Dec;                                // SP--
    mar_br sp_bw;                          // MAR <-- SP
    ram_br pc_bw;                          // RAM[mar] <-- PC
    pc_br i_bw;                            // PC <-- I
    ctrl_start ctrl_load;                  // CNT <-- 0
//
return:
    mar_br sp_bw;                          // MAR <-- SP
    sp_Inc;                                // SP++;
    pc_br ram_bw;                          // PC <-- RAM[mar]
    ctrl_start ctrl_load;                  // CNT <-- 0
//
push_mdr:
    sp_Dec;                                // SP--
    mar_br sp_bw;                          // MAR <-- SP
    ram_br mdr_bw;                         // RAM[mar] <-- MDR
    ctrl_start ctrl_load;                  // CNT <-- 0
//
push_a:
    sp_Dec;                                // SP--
    mar_br sp_bw;                          // MAR <-- SP
    ram_br a_bw;                           // RAM[mar] <-- A
```

```
    ctrl_start ctrl_load;               // CNT <-- 0
//
push_b:
    sp_Dec;                             // SP--
    mar_br sp_bw;                       // MAR <-- SP
    ram_br b_bw;                        // RAM[mar] <-- B
    ctrl_start ctrl_load;               // CNT <-- 0
//
push_fl:
    sp_Dec;                             // SP--
    mar_br sp_bw;                       // MAR <-- SP
    ram_br fl_bw;                       // RAM[mar] <-- FL
    ctrl_start ctrl_load;               // CNT <-- 0
//
push_i:
    sp_Dec;                             // SP--
    mar_br sp_bw;                       // MAR <-- SP
    ram_br i_bw;                        // RAM[mar] <-- I
    ctrl_start ctrl_load;               // CNT <-- 0
//
pop_mdr:
    mar_br sp_bw;                       // MAR <-- SP
    sp_Inc;                             // SP++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    ctrl_start ctrl_load;               // CNT <-- 0
//
pop_a:
    mar_br sp_bw;                       // MAR <-- SP
    sp_Inc;                             // SP++
    a_br ram_bw;                        // A <-- RAM[mar]
    ctrl_start ctrl_load;               // CNT <-- 0
//
pop_b:
    mar_br sp_bw;                       // MAR <-- SP
    sp_Inc;                             // SP++
    b_br ram_bw;                        // B <-- RAM[mar]
    ctrl_start ctrl_load;               // CNT <-- 0
//
pop_fl:
    mar_br sp_bw;                       // MAR <-- SP
    sp_Inc;                             // SP++
```

```
    fl_br ram_bw;                    // FL <-- RAM[mar]
    ctrl_start ctrl_load;            // CNT <-- 0
//
pop_i:
    mar_br sp_bw;                    // MAR <-- SP
    sp_Inc;                          // SP++
    i_br ram_bw;                     // I <-- RAM[mar]
    ctrl_start ctrl_load;            // CNT <-- 0
…
end
```

Figure 8.130. Match between the contents of the memory that represents the microcode (the couple of memory banks **m1** and **m2** inside the control unit) and the microcode description itself.




stop
i_bw
i_br
b_bw
b_br
fl_bw
fl_br
fl_ar
ir_bw
ir_br
ram_bw
ram_br
mar_bw
mar_br
mdr_bw
mdr_br
pc_Inc
pc_bw
pc_br
ctrl_load
ctrl_start
b35 b34 b33 b32 b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
push_a:
push_b:
push_fl:
push_i:
sp_Dec
sp_Inc
sp_bw
sp_br
alu_bw
0=not A
1=A and B
2=A or B
3=A xor B
4=A << 1
5=A >> 1
6=A << 1 arith
7=A >> 1 arith
8=A rot left
9=A rot right
A=A rot carry left
B=A rot carry right
C=A plus B carry
D=A minus B borrow
E=A plus B
F=A minus B
a_bw
a_br
sel_bw
0=if carry
1=if not carry
2=if zero
3=if not zero
4=if negative
5=if not negative
6=if overflow
7=if not overflow

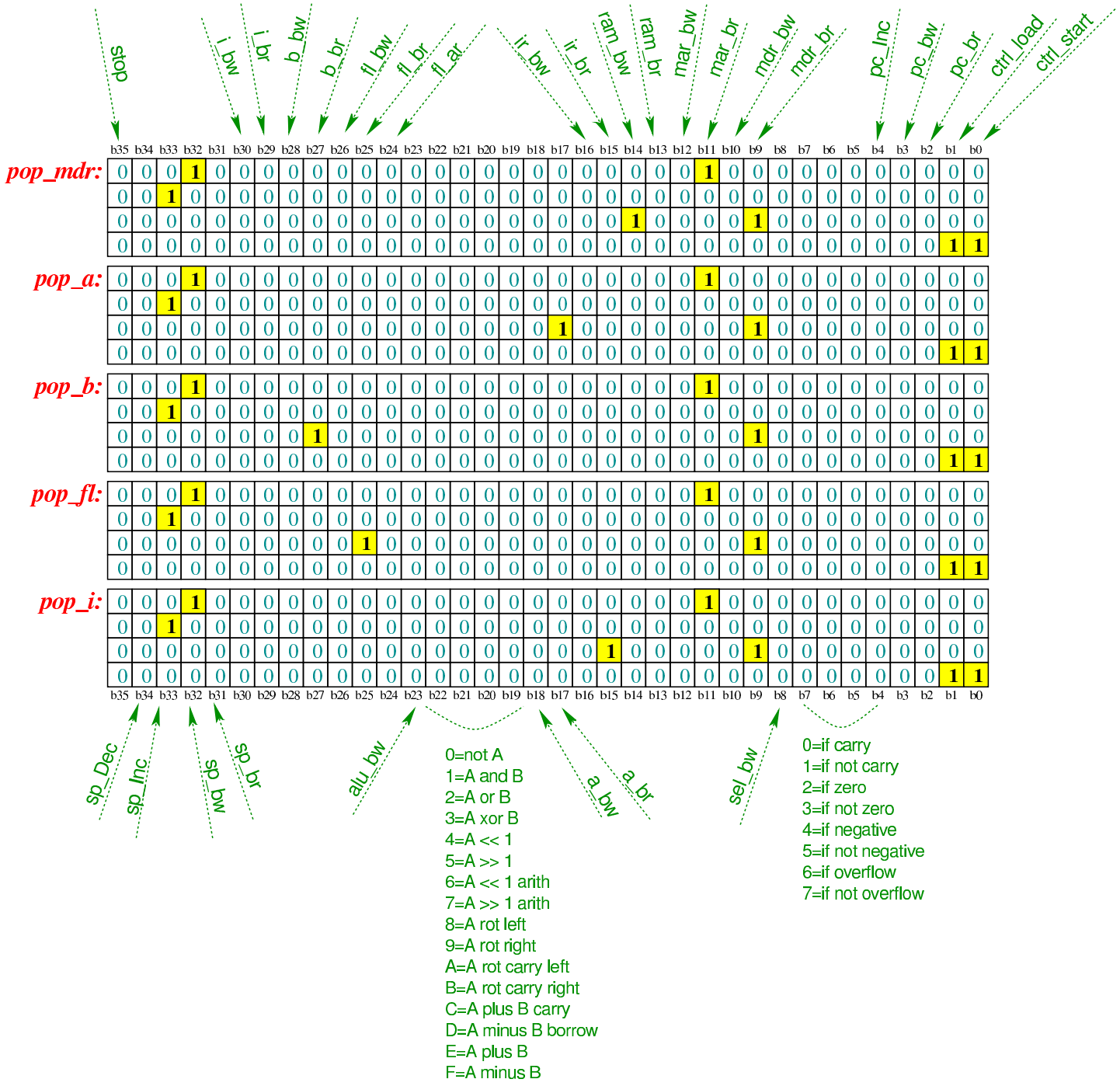

stop
i_bw
i_br
b_bw
b_br
fl_bw
fl_br
fl_ar
ir_bw
ir_br
ram_bw
ram_br
mar_bw
mar_br
mdr_bw
mdr_br
pc_Inc
pc_bw
pc_br
ctrl_load
ctrl_start
b35 b34 b33 b32 b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
pop_mdr:
pop_a:
pop_b:
pop_fl:
pop_i:
sp_Dec
sp_Inc
sp_bw
sp_br
alu_bw
0=not A
1=A and B
2=A or B
3=A xor B
4=A << 1
5=A >> 1
6=A << 1 arith
7=A >> 1 arith
8=A rot left
9=A rot right
A=A rot carry left
B=A rot carry right
C=A plus B carry
D=A minus B borrow
E=A plus B
F=A minus B
a_bw
a_br
sel_bw
0=if carry
1=if not carry
2=if zero
3=if not zero
4=if negative
5=if not negative
6=if overflow
7=if not overflow

Table 8.131. Macroinstructions added for this version of the simple CPU.

Syntax	Description
`call_ref #`***ref***	It decrements the value inside the ***SP*** register and save in memory, at the new ***SP*** position, the address of the next instruction (after **`call`**_...), then it updates the ***PC*** register with the address specified (the argument). That is: it calls the procedure storing inside the stack the return address.
`call_reg`	It decrements the value inside the ***SP*** register and save in memory, at the new ***SP*** position, the address of the next instruction (after **`call`**_...), then it updates the ***PC*** register with the address contained inside ***I***. That is: it calls the procedure storing inside the stack the return address.
`return`	It increments the ***SP*** register and it restore the register ***PC*** loading from memory the content at the address pointed by the updated ***SP***. That is: it returns from a procedure call.
`push_mdr`	It saves on top of the stack the value of ***MDR***.
`push_a`	It saves on top of the stack the value of ***A***.
`push_b`	It saves on top of the stack the value of ***B***.
`push_fl`	It saves on top of the stack the value of ***FL***.
`push_i`	It saves on top of the stack the value of ***I***.
`pop_mdr`	It pops from the stack a value that is loaded by the ***MDR*** register.
`pop_a`	It pops from the stack a value that is loaded by the ***A*** register.
`pop_b`	It pops from the stack a value that is loaded by the ***B*** register.
`pop_fl`	It pops from the stack a value that is loaded by the ***FL*** register.

Syntax	Description
`pop_i`	It pops from the stack a value that is loaded by the ***I*** register.

8.7.1 Instruction "push" and "pop"

«

Listing 8.132. Macrocode example to experiment the instruction for adding and removing values from the stack: it is saved a value from the ***A*** register and then it is restored inside the ***B*** register. The complete source file should be available at *attachments/xcpu/xcpu-a7-push-pop.gm* .

```
begin macrocode @ 0
start:
        load_ref #sp_bottom
        move_mdr_sp
        load_ref #data_0
        move_mdr_a
        push_a
        pop_b

stop:
        stop
sp_bottom:
        .byte 0x10
data_0:
        .byte 0xCC
end
```

Figure 8.133. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value. The memory cell at the address $0F_{16}$ is written by the instruction **`push_a`**.

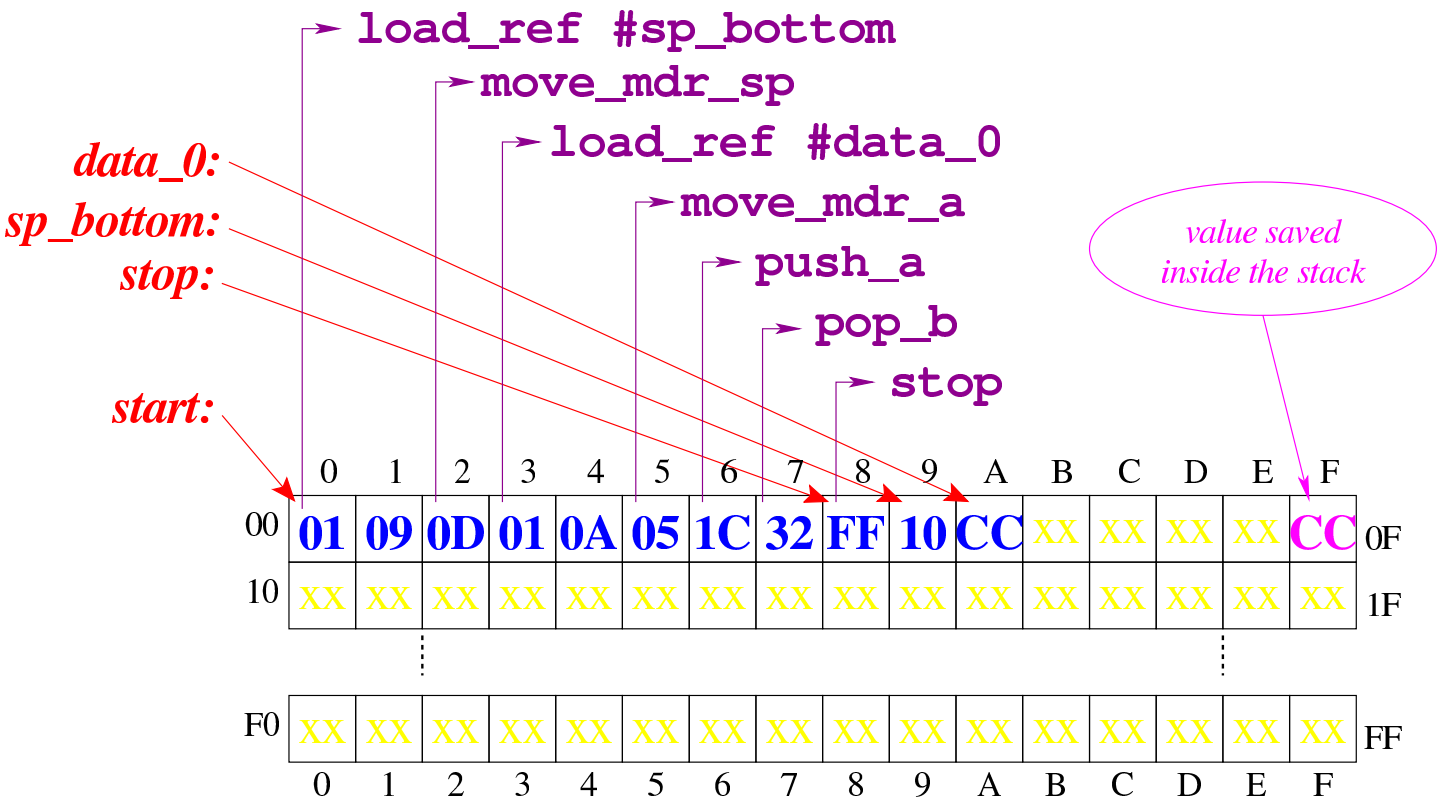
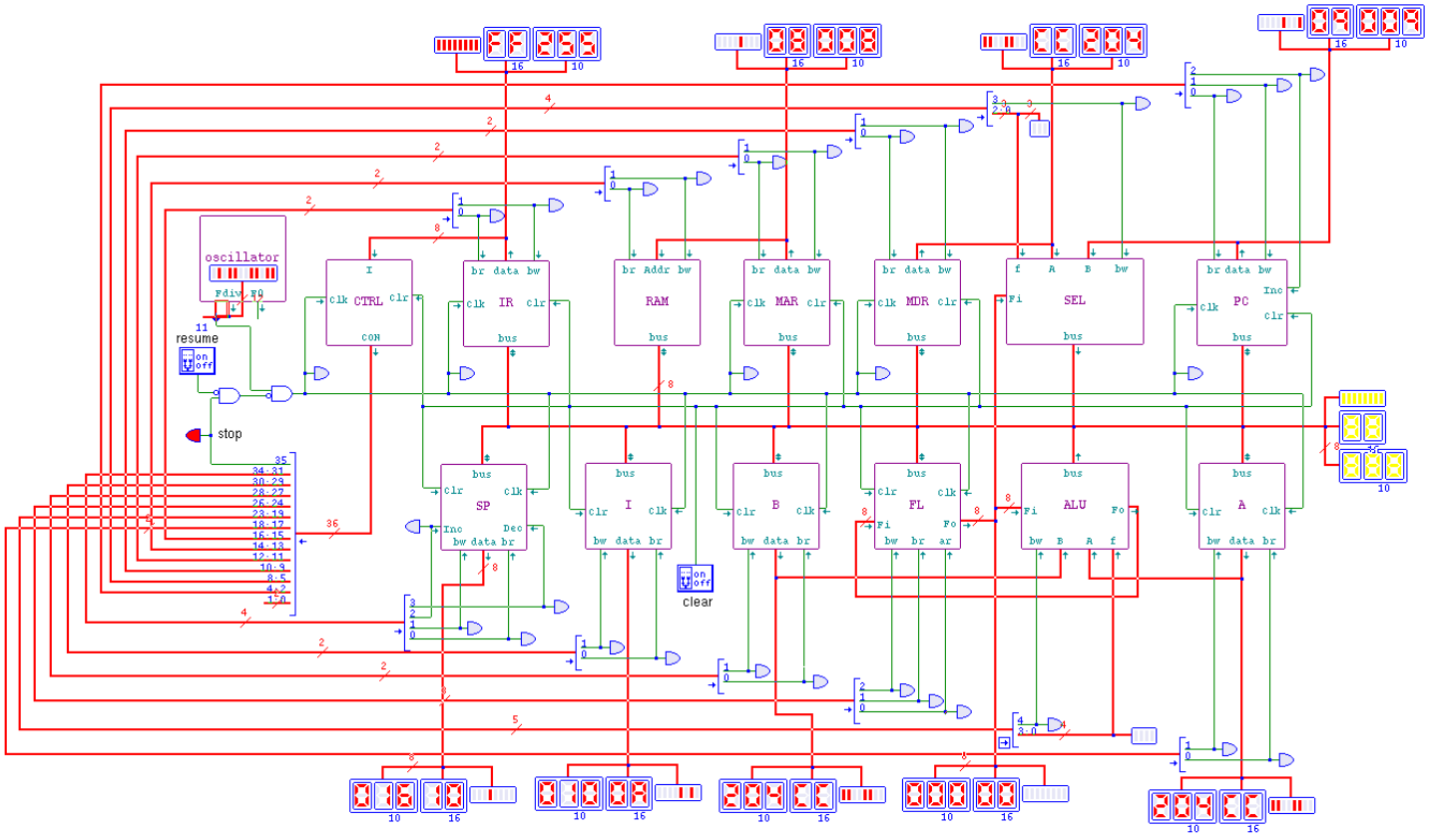


Figure 8.134. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=vAdVww3lD7I*



8.7.2 Instruction "call" and "return"

Listing 8.135. Macrocode example to experiment the instructions for calling a procedure and returning from a procedure. The complete source file should be available at *attachments/xcpu/xcpu-a7-call-return.gm* .

```
begin macrocode @ 0
start:
        load_ref #sp_bottom
        move_mdr_sp
        call_ref #elab
        move_a_mdr
        move_mdr_b
        jump #stop
elab:
        load_ref #data_0
        move_mdr_a
```

```
        load_ref #data_1
        move_mdr_b
        add
        return
stop:
        stop
sp_bottom:
        .byte 0x20
data_0:
        .byte 0x0A
data_1:
        .byte 0x0B
end
```

Figure 8.136. RAM memory content: the memory cells marked with "xx" have an unknown and unspecified value. The memory cell at the address $1F_{16}$ is written by the instruction **`call_ref`**.

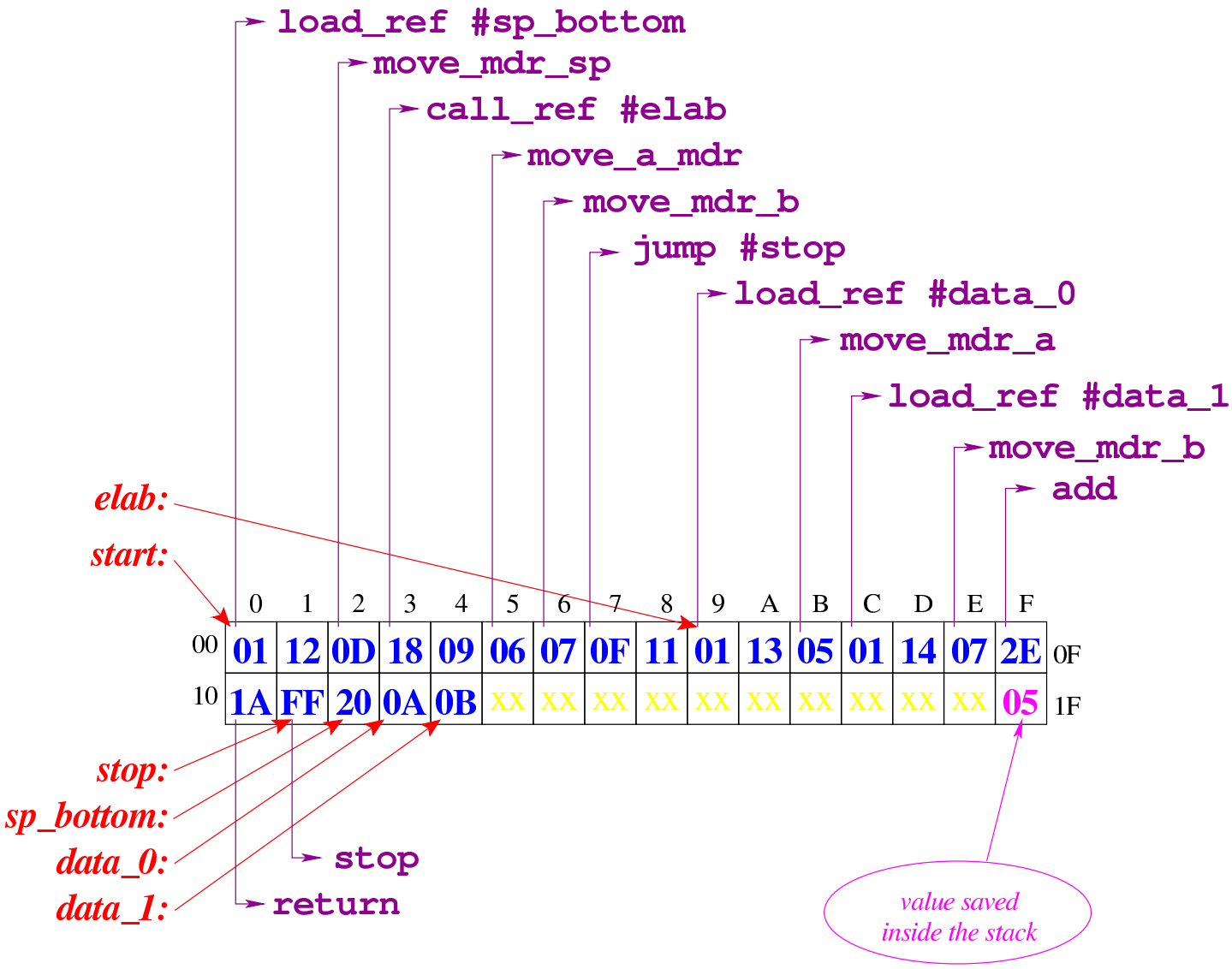
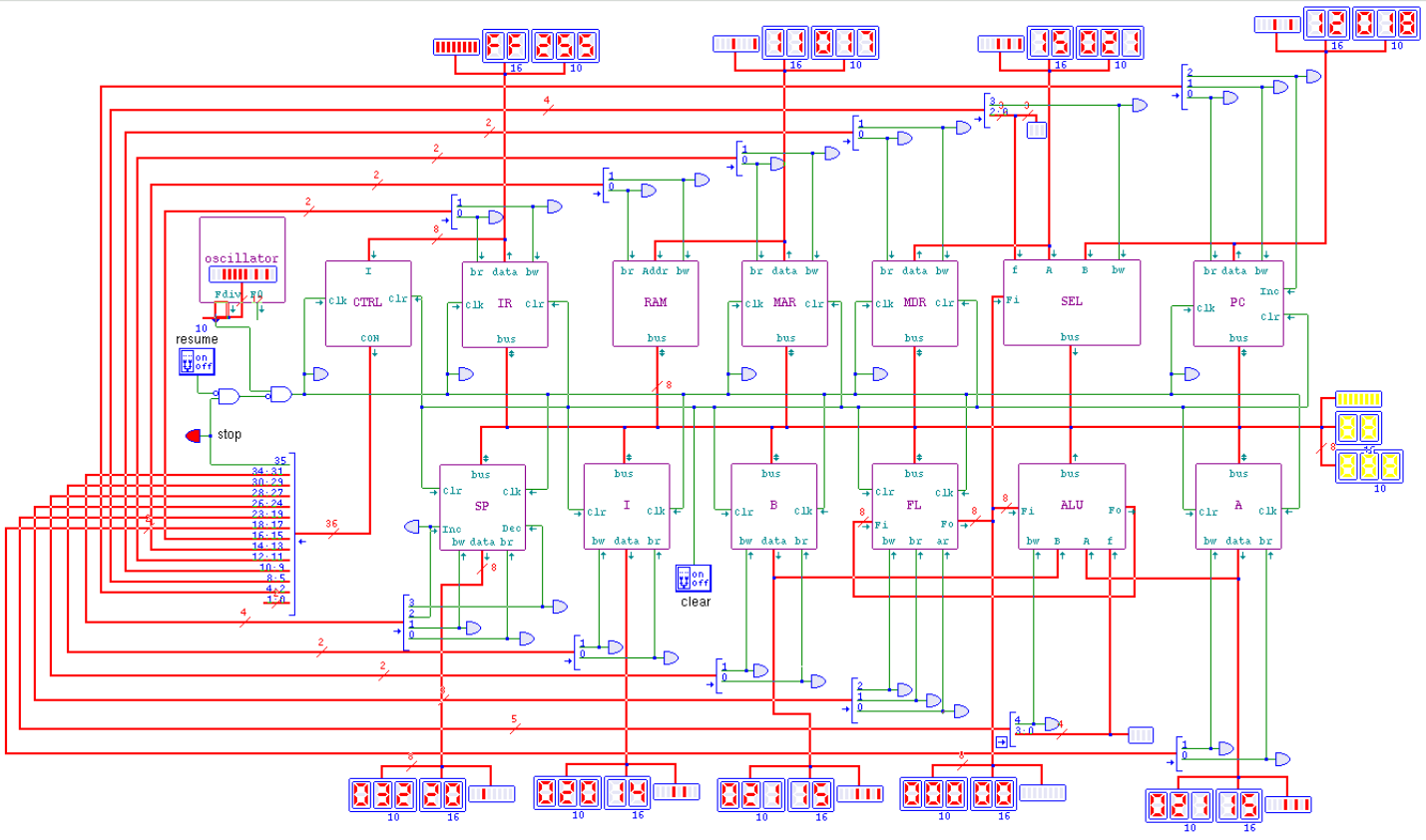


Figure 8.137. Data bus after the execution of the above example. Video: ogv *http://www.youtube.com/watch?v=nWdXMvegkjc*



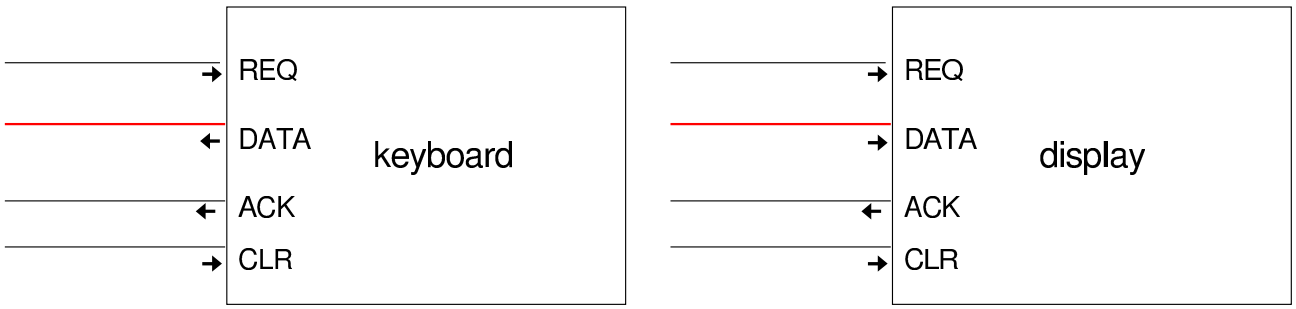
8.8 Version "A8": I/O

The previous version of the simple CPU is complete enough to show the main functions of a common CPU. This version extends the project introducing I/O devices. Two new busses are added, used for the data to move with devices and to select a specific device.

8.8.1 Communication with asynchronous devices

Usually, the I/O devices are not synchronous with the CPU clock, therefore it is necessary a protocol, so that it can be made a request of an action to the device and a feedback from the device can be received. The I/O devices used in this simple CPU project have externally the connections shown in the next figure.

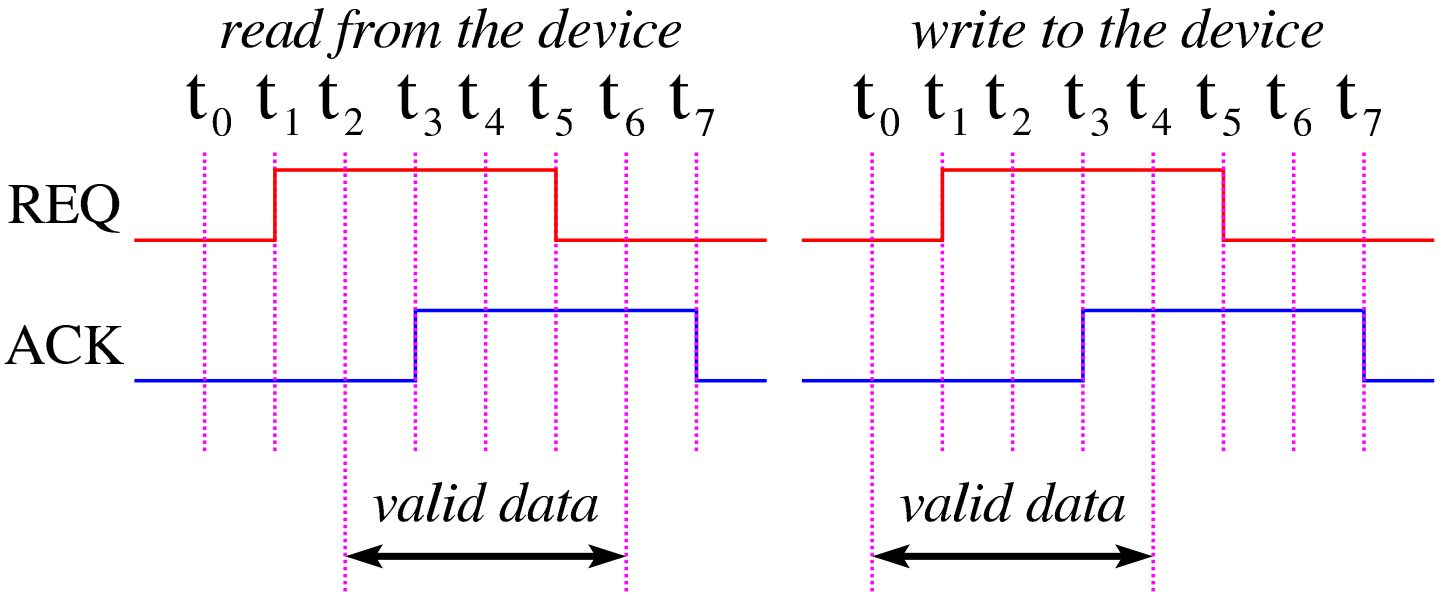
Figure 8.138. One input device and one output device.



The input or output ***DATA*** allows to move the information that is needed by the device or that is generated from it. The inputs ***REQ*** (request) and ***ACK*** (acknowledge) are used to negotiate the information movement (the input ***CLR*** is used only to clear the device). Two examples should be done, to examine the procedure of reading from the device and of writing on it.

To communicate with an **input device**, from which a data is to be read, it is required to activate the input ***REQ*** (t_1) to request the data. The device receives the request and prepares the output ***DATA*** with the information (t_2), then it activates the output ***ACK*** (t_3). At that point, finding the output ***ACK*** active, it is possible to read from the output ***DATA*** (t_4) and after the read operation it is possible to disable the input ***REQ*** (t_5). The device, finding the input ***REQ*** disabled, knows that the information was taken, then it stops to keep it (t_6) and disables the output ***ACK*** (t_7).

Figure 8.139. Phases for reading an input device ad for writing to an output device.



To communicate with an output device, where an information is to be written, it is

necessary to prepare that data to the input ***DATA*** (t_0), then it is necessary to activate the input ***REQ*** (t_1) to tell the device that the data is ready. Then the device gets the information (t_2) and it confirms activating the output ***ACK*** (t_3). Finding the confirmation, it is not necessary to keep the information on the input ***DATA*** ***DATA*** (t_4) and the input ***REQ*** is disabled (t_5). The device, finding that the request is completed, disables the output ***ACK*** (t_7) concluding the operation.

8.8.2 I/O devices implementation

At the moment, only two I/O devices are implemented: a keyboard and a screen. As the project is developed with TKGate, the devices are made through TCL/Tk code. The code that is shown was obtained modifying similar devices from the TKGate library.

Listing 8.140. File '`share/tkgate/vpd/kbd.tcl`' to simulate the keyboard input. The program shows a little empty window, which should be selected to type something that could be received by the related virtual device.

```
VPD::register KBD
VPD::allow KBD::post

namespace eval KBD {
    # Public variables declaration. $kbd_w is an array that is
    # use only for the element $n. $kbd_w($n) identifies uniquely
    # the working interface instance.
    variable kbd_w
    variable KD
    # Function required by TKGate to create the interface.
    proc post {n} {
        variable kbd_w
        # It creates a window and saves the object inside the array
        # $kbd_w.
        set kbd_w($n) [VPD::createWindow "KBD $n" -shutdowncommand "KBD::unpost $n"]
        # It links the keyboard typing, related to the object represented
        # by $kbd_w($n), to the sendChar function.
        bind $kbd_w($n) <KeyPress> "KBD::sendChar $n \"%A\""
        # It opens a write channel, named "KD".
        if {[info exists ::tkgate_isInitialized]} {
            VPD::outsignal $n.KD KBD::KD($n)
        }
    }
    # Function that gets the typing and puts it into the channel
    # "KD", related to the current interface instance.
    proc sendChar {n key} {
```

```
        variable KD
        if { [string length $key ] == 1 } {
            binary scan $key c c
            set KBD::KD($n) $c
        }
    }
    # Function required by TKGate to destroy the interface.
    proc unpost {n} {
        variable kbd_w
        destroy $kbd_w($n)
        unset kbd_w($n)
    }
}
```

Listing 8.141. File 'share/tkgate/vpd/scr.tcl' to simulate the output on a character screen. The program shows a window where the sent characters are shown. Inside the code there is reference to the file 'textcurs.b' that is already included with the TKGate distribution.

```
image create bitmap txtcurs -file "$bd/txtcurs.b"

VPD::register SCR
VPD::allow SCR::post
VPD::allow SCR::data

namespace eval SCR {
    # Public variables declaration: they are arrays where only
    # the element $n is used. The element $n identify uniquely
    # the working interface instance.
    variable scr_w
    variable scr_pos
    # Function required by TKGate to create the interface.
    proc post {n} {
        variable scr_w
        variable scr_pos
        # It creates the window and it saves the object inside the array
        # $scr_w.
        set scr_w($n) [VPD::createWindow "SCR $n" -shutdowncommand "SCR::unpost $n"]
        # For convenience, it copies the object reference inside the local variable
        # $W and then the object is referred only with that variable.
        set w $scr_w($n)
        text $w.txt -state disabled
        pack $w.txt
        # It puts the cursor at the end of the shown text.
        $w.txt image create end -image txtcurs
```

```
        # It opens a reading channel, called "RD" and it connects it to the
        # function "data".
        if {[info exists ::tkgate_isInitialized]} {
            VPD::insignal $n.RD -command "SCR::data $n" -format %d
        }
        # It clears the counter for the shown characters on screen.
        set scr_pos($n) 0
    }
    # Function required by TKGate to destroy the interface.
    proc unpost {n} {
        variable scr_w
        destroy $scr_w($n)
        unset scr_w($n)
    }
    # Function used for receiving the data to be shown on screen.
    proc data {n c} {
        variable scr_w
        variable scr_pos
        # For convenience, it copies the interface object reference inside
        # the local variable $w.
        set w $scr_w($n)
        catch {
            # The $c variable contains the character to be shown.
            if { $c == 7 } {
                # BEL
                bell
                return
            } elseif { $c == 127 || $c == 8 } {
                # DEL | BS
                if { $scr_pos($n) > 0 } {
                    # It deletes the last shown character, but only if the
                    # characters counter is greater then zero, otherwise
                    # the cursor would disappear and the newly received
                    # characters could be placed inside an invisible area
                    # of the virtual screen.
                    $w.txt configure -state normal
                    $w.txt delete "end - 3 chars"
                    $w.txt see end
                    $w.txt configure -state disabled
                    set scr_pos($n) [expr {$scr_pos($n) - 1}]
                }
                return
            } elseif { $c == 13 } {
                # CR is translated into LF.
                set c 10
            }
```

```
            # It converts the character number into a symbol that can
            # be shown.
            set x [format %c $c]
            # It shows the symbol.
            $w.txt configure -state normal
            $w.txt insert "end - 2 chars" $x
            $w.txt see end
            $w.txt configure -state disabled
            # It updates the shown characters counter.
            set scr_pos($n) [expr {$scr_pos($n) + 1}]
        }
    }
}
```

The two TCL/Tk programs are called by some Verilog code. The following listings show the modules **keyboard** and **display**, which appear graphically as in the figure 8.138.

Listing 8.142. Verilog code for the **keyboard** module.

```
module keyboard(DATA, REQ, ACK, CLR);
output ACK;
output [7:0] DATA;
input REQ;
input CLR;
reg ready;
reg [7:0] key;

  initial
    begin
      ready = 0;
      key = 0;
    end

  always
    begin
      @(posedge CLR)
      ready = 0;
      key = 0;
```

```
    end

  initial $tkg$post("KBD","%m");

  always
    begin
      @ (posedge REQ);
      # 5;
      key = $tkg$recv("%m.KD");
      # 5;
      ready = 1'b1;
      # 5;
      @ (negedge REQ);
      # 5;
      ready = 1'b0;
    end

  assign DATA = key;
  assign ACK = ready;

endmodule
```

Listing 8.143. Verilog code for the **`display`** module.

```
module display(DATA, REQ, ACK, CLR);
output ACK;
input [7:0] DATA;
input REQ;
input CLR;
reg ready;

  initial
    begin
      ready = 0;
    end
```

```
  initial $tkg$post("SCR","%m");

  always
    begin
      @(posedge CLR)
      ready = 0;
    end

  always
    begin
      @(posedge REQ);
      # 5;
      $tkg$send("%m.RD",DATA);
      # 5;
      ready = 1'b1;
      # 5;
      @(negedge REQ);
      # 5;
      ready = 1'b0;
    end

  assign ACK = ready;

endmodule
```

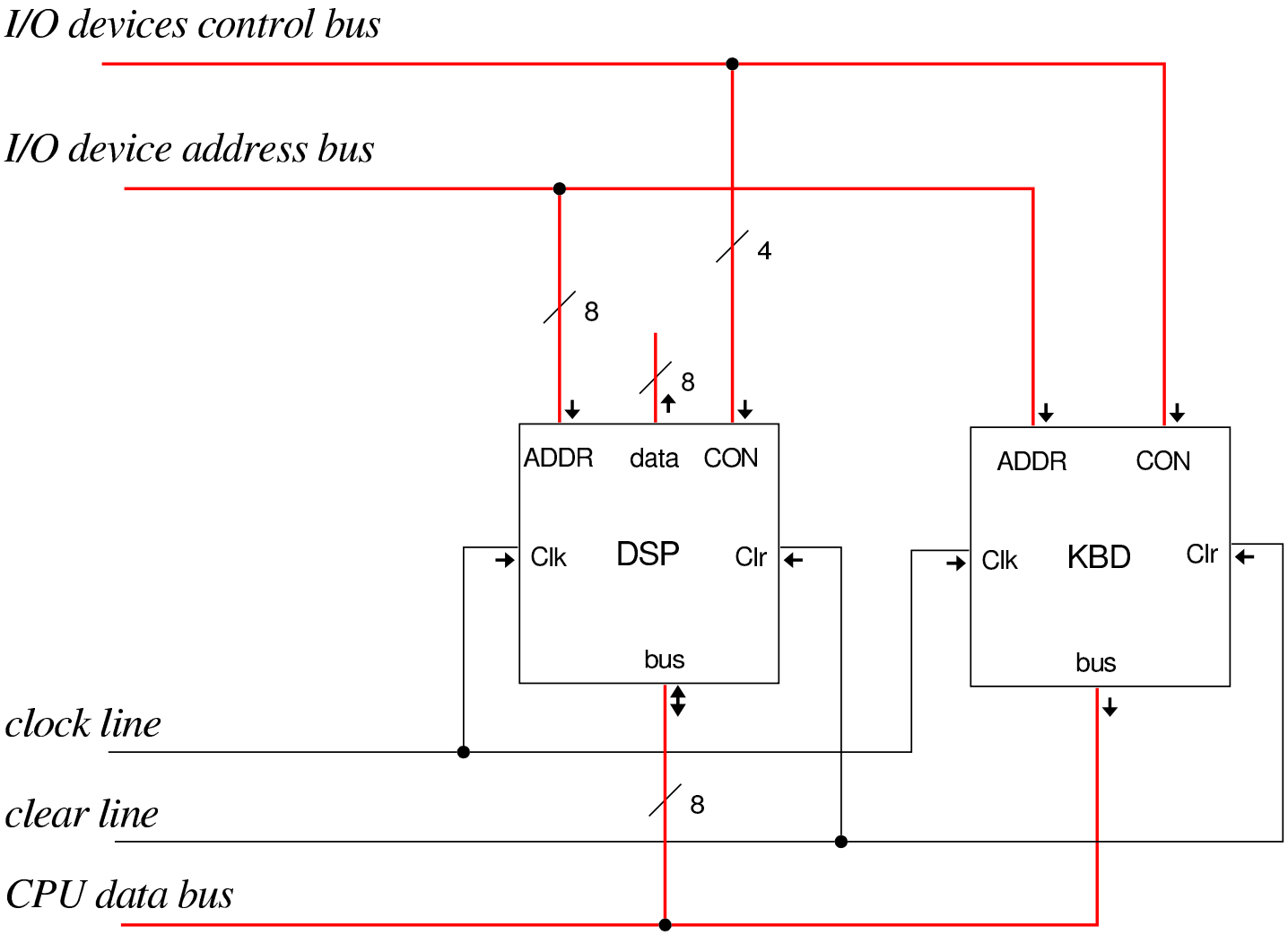
Until now, the described devices can work only asynchronously, but to be used, they must be adapted to work synchronously with the CPU.

8.8.3 External appearance of the synchronous interfaces

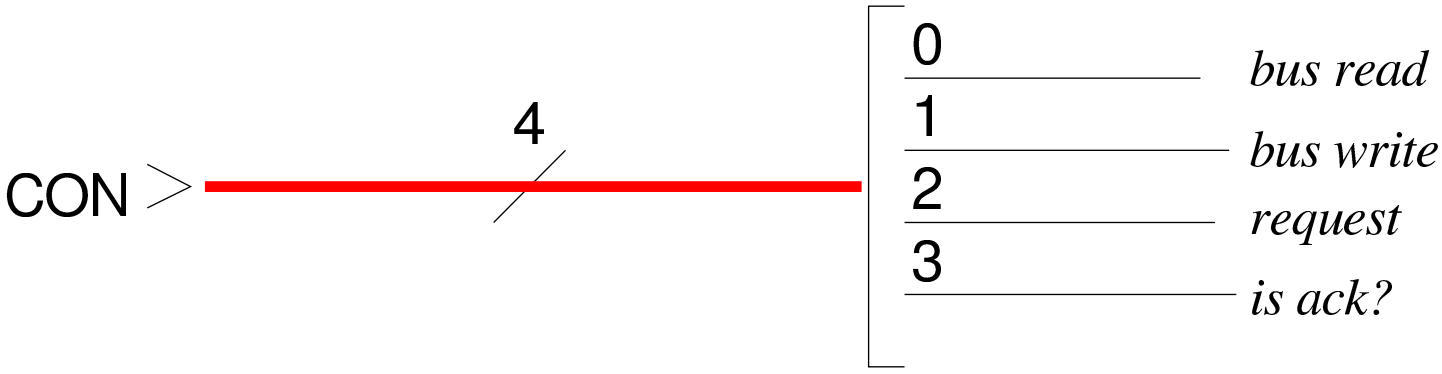
«

The synchronous interfaces for the I/O devices should be connected to the CPU data bus, just like the other modules already implemented. However, to simplify the connections, instead of having personal control lines, an additional bus is created for the I/O module address to be selected.

Figure 8.144. I/O modules connection.



As the figure shows, the bus connected to the inputs ***ADDR*** is used to select the device, while the bus connected to the inputs ***CON*** is used to uniform the control lines for all the I/O modules. Please notice that the inputs ***CON*** are the same for all I/O modules, but every module uses only the lines that it requires, ignoring the others:



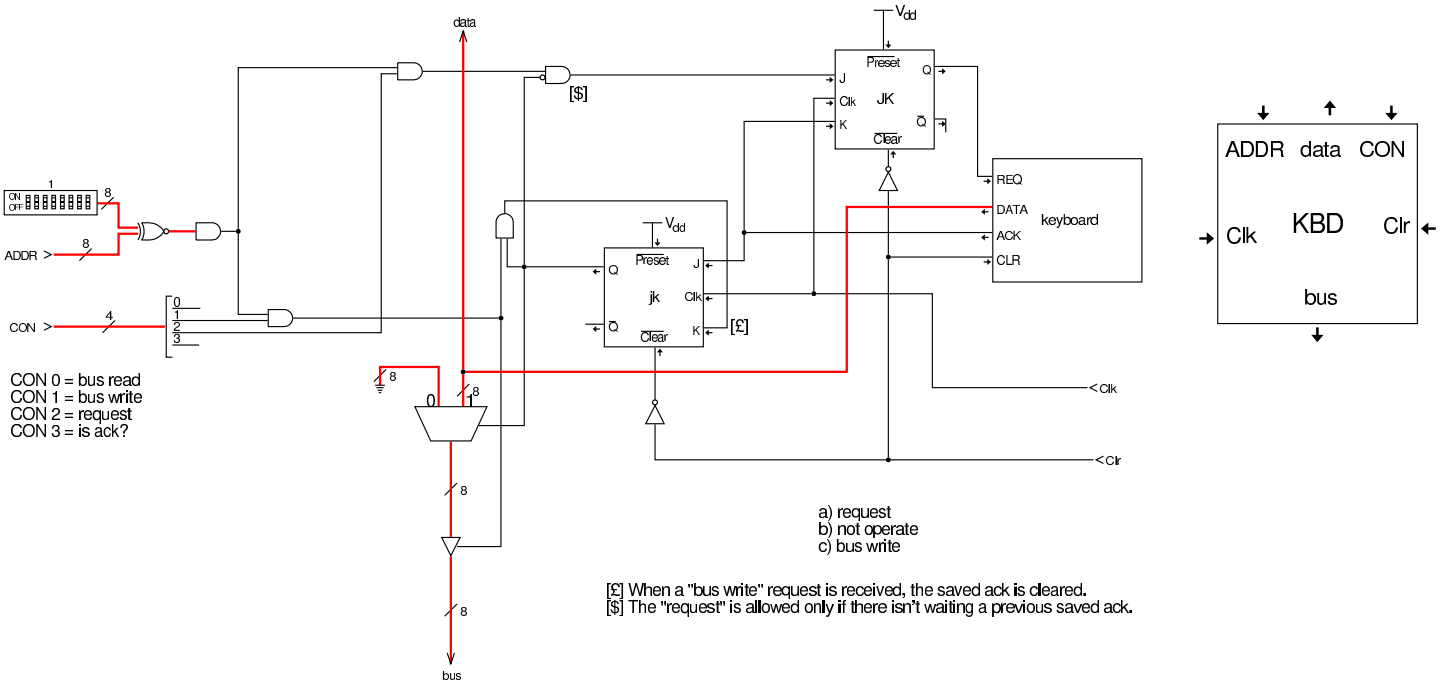
The lines ***bus read*** and ***bus write*** are the same of the other modules already described

and they refer to the request to read the CPU data bus or write to the CPU data bus. The line ***request*** is used to request a read or write operation on the device, but without the need to keep it active as it should for the asynchronous protocol already mentioned. The line ***is ack?*** is used to obtain, somehow, an acknowledgment from the device.

8.8.4 Synchronous keyboard interface

The keyboard interface module uses only two out of the four control lines: ***bus write*** and ***request***.

Figure 8.146. Module **KBD**, which connects the keyboard to the CPU data bus.



To communicate with the keyboard module it is necessary to start with a ***request*** signal. Then, after a positive clock edge it is saved inside the upper JK flip-flop, which is activated. That flip-flop has the task to keep the request active on the input ***REQ*** of the device. The second central JK flip-flop has the task to save the acknowledge produced by the output ***ACK*** and, if that acknowledge is active inside the flip-flop, it does not allow to receive another request. After a ***request*** is correctly received and saved, if the device has a character ready to be output, it replies almost immediately, placing that character on its ***DATA*** output and activating the output ***ACK***. At the second positive clock edge, the flip-flop responsible for the request signal is cleared and instead is activated the second flip-flop responsible for the acknowledge. However, the device continue to keep its ***DATA*** with the character to be read. Then, a ***bus write*** signal is received, allowing to write the character read from the device into the CPU

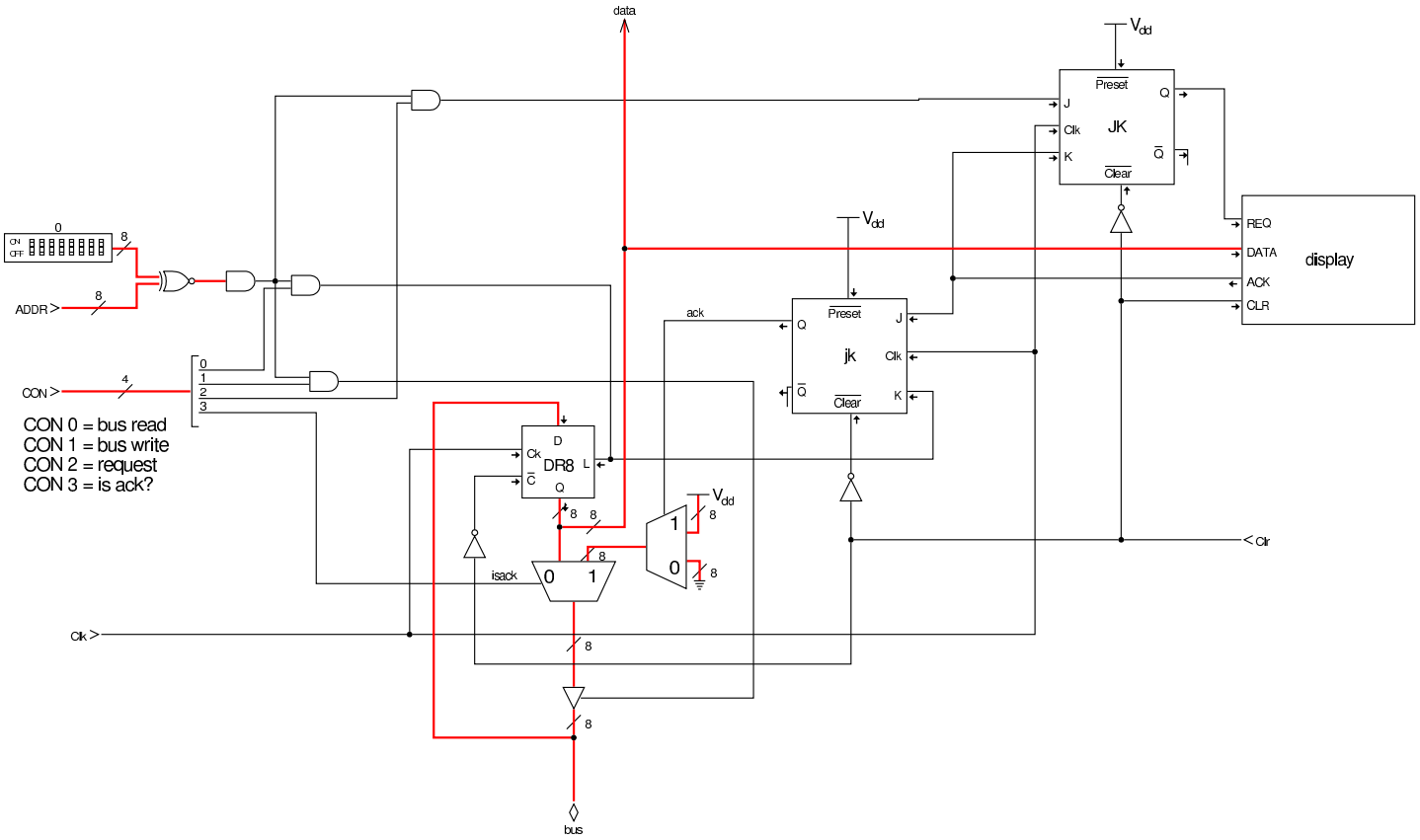
data bus. This action clears also the acknowledge flip-flop. If otherwise, the keyboard device is not yet ready to offer a character, the interface will write to the CPU data bus a null value (00_{16}), because the acknowledge flip-flop is not activated.

At the macrocode level, if the keyboard read produces a null character, it just means that there is not a character ready and the keyboard read should be retried.

8.8.5 Synchronous display interface

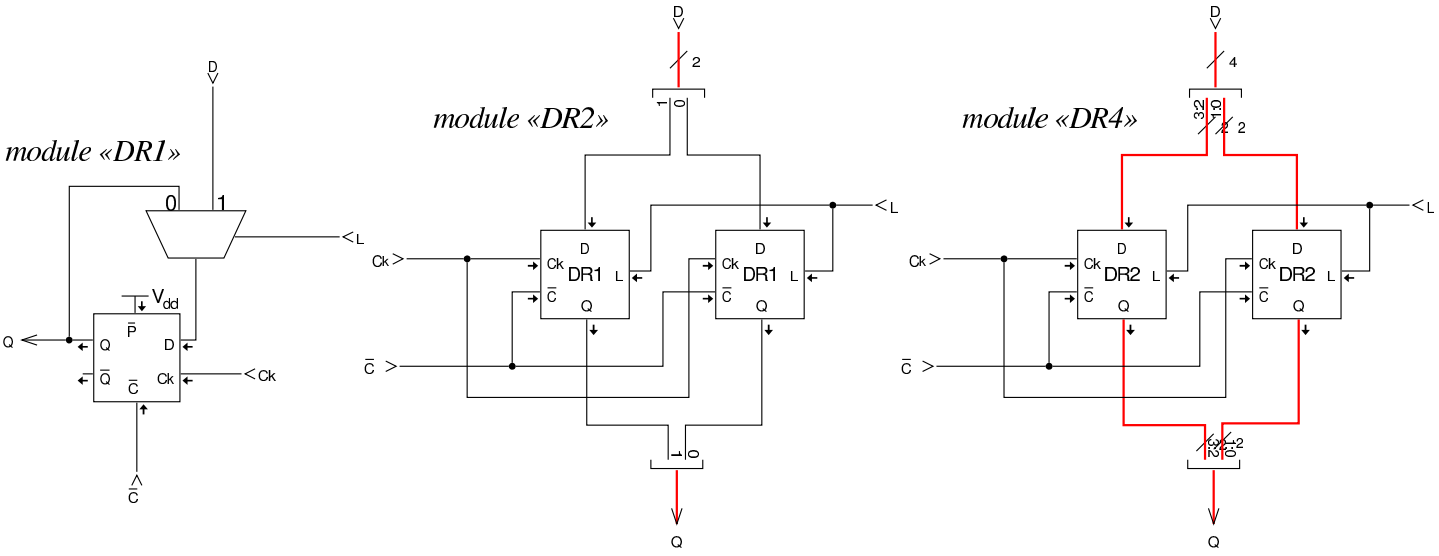
The **DSP** module uses all the control lines, because it must be able to read from the CPU data bus when a character is to be shown on the screen, but must also be able to write to the bus to give the acknowledge of the work properly done.

Figure 8.147. Module **DSP**.



The **DSP** module uses a register made out of eight D flip-flops (positive edge triggered, module **DR8**) which is used to save the character to be shown on the screen. However, when the control line ***is ack?*** is active, the **DSP** module writes to the CPU data bus the result of its work: 00_{16} means that the last character to be displayed is not already appeared, whereas FF_{16} means that the last character to be displayed was correctly shown.

Figure 8.148. Building **DR*n*** modules.



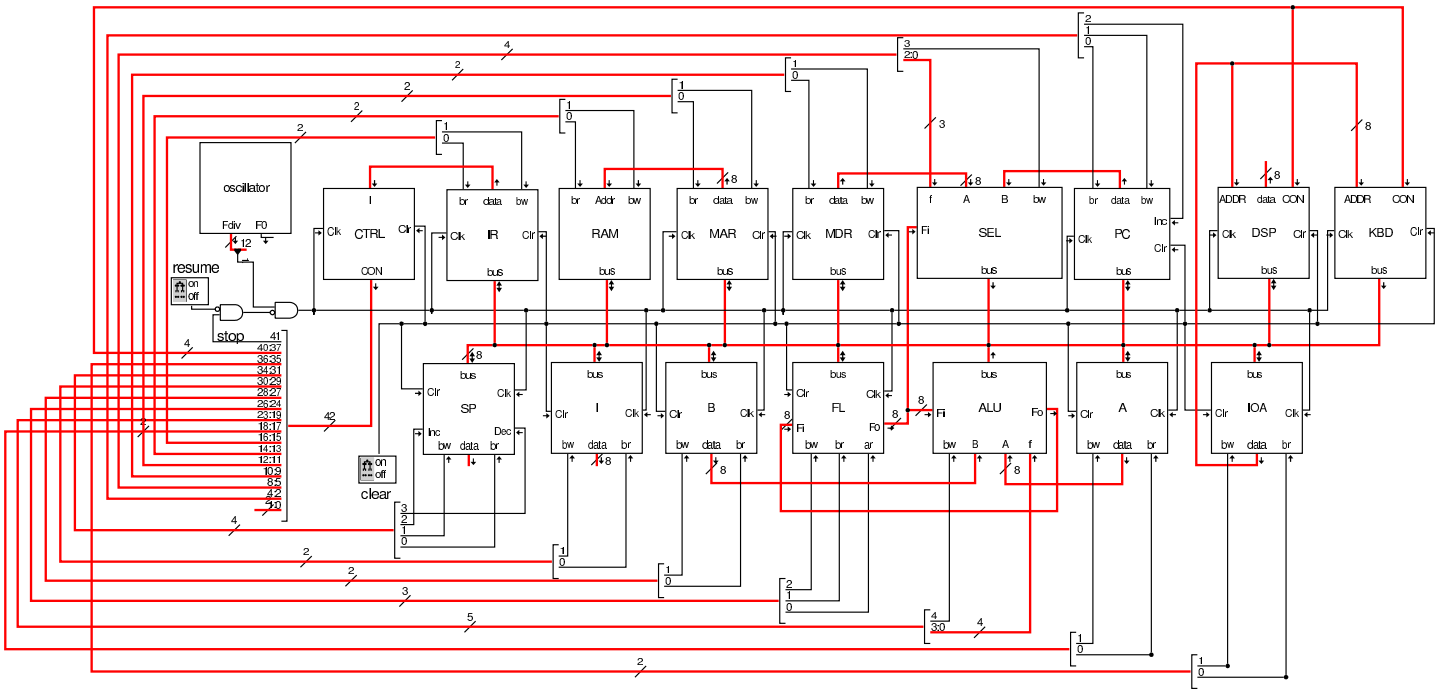
To show a character on screen, it is necessary to start activating the control line ***bus read***: after the next clock positive edge the internal data register loads the character reading it from the CPU data bus, while the JK flip-flop at the middle is cleared and so is the ***ack*** multiplexer, connected to the flip-flop output. Then the control line ***request*** should be activated: after the next clock positive edge this value is saved inside the JK flip-flop on the top side, activating the input ***REQ*** of the device. Then, after a while, the device shows the character that is already available at its ***DATA*** input (it comes from the data register) and then it replies activating its ***ACK*** output. When the ***ACK*** output from the device is active and a new clock positive edge comes, the JK flip-flop on the top is reset and the other JK flip-flop is activated, switching the connected multiplexer.

To show a character, only two clock cycles are required, but to be able to verify if there is an acknowledge, it is necessary to read the device after it. To check the acknowledgement it is necessary to activate the control line ***is ack?*** and ***bus write***, so that the interface can write to the bus the value ***00*** for a negative acknowledge, or ***FF*** for a positive acknowledge.

8.8.6 The CPU data bus with the I/O interfaces

Together with the I/O interface modules, also a new register is added: ***IOA*** (*I/O address*). When a device is to be reached, it is necessary to load its I/O address into the ***IOA*** register, before anything else can be done with the device itself.

Figure 8.149. The CPU data bus with the ***IOA*** register and the I/O interfaces.



The CPU control bus is now modified with the addition of the lines used to control the ***IOA*** register and the I/O interfaces. That modification has required to shift the line used to stop the clock. Therefore, inside the memory banks declaration code and inside the control word fields declaration code there are the following adaptations:

```
map        bank[7:0]    ctrl.m0;
microcode  bank[31:0]   ctrl.m1;
microcode  bank[41:32]  ctrl.m2;
macrocode  bank[7:0]    ram.m3;
…
field ioa_br[35];          // IOA <-- bus
field ioa_bw[36];          // IOA --> bus
field io_br[37];           // I/O <-- bus
field io_bw[38];           // I/O --> bus
field io_req[39];          // I/O request
field io_isack[40];        // I/O is ack?
field stop[41];            // stop clock
```

Inside the opcode list there are new instructions and the same is for the microcode description. There is also a new instruction that requires two arguments:

```
operands op_2 {
    //
    // [........][mmmmmmmm][nnnnnnnn]
    //
    #1,#2 = { +1=#1[7:0]; +2=#2[15:8]; };
};
…
op in {
  map in : 48;                    // read input from I/O bus
  +0[7:0]=48;
  operands op_1;
};
op out {
  map out : 49;                   // write output to I/O bus
  +0[7:0]=49;
  operands op_1;
};
op io_is_ack {
  map io_is_ack : 50;
  +0[7:0]=50;
  operands op_2;
};
```

```
begin microcode @ 0
…
in:
    mar_br pc_bw;               // MAR <-- PC
    pc_Inc;                     // PC++
    mdr_br ram_bw;              // MDR <-- RAM[mar]
    ioa_br mdr_bw;              // IOA <-- MDR
    io_req;                     //
    io_br;                      // does not do anything
    a_br io_bw;                 // A <-- I/O
    ctrl_start ctrl_load;       // CNT <-- 0

out:
    mar_br pc_bw;               // MAR <-- PC
    pc_Inc;                     // PC++
    mdr_br ram_bw;              // MDR <-- RAM[mar]
```

```
    ioa_br mdr_bw;                  // IOA <-- MDR
    io_br a_bw;                     // IO <-- A
    io_req;                         //
    ctrl_start ctrl_load;           // CNT <-- 0

io_is_ack:
    mar_br pc_bw;                   // MAR <-- PC
    pc_Inc;                         // PC++
    mdr_br ram_bw;                  // MDR <-- RAM[mar]
    ioa_br mdr_bw;                  // IOA <-- MDR
    //
    mar_br pc_bw;                           // MAR <-- PC
    pc_Inc;                                 // PC++
    mdr_br ram_bw;                          // MDR <-- RAM[mar]
    a_br io_bw io_isack;                    // A <-- I/O is ack
    a_br alu_f=not_a alu_bw fl_ar;          // A <-- NOT A
    a_br alu_f=not_a alu_bw fl_ar;          // A <-- NOT A
    pc_br sel_f=if_not_zero sel_bw          // PC = (not_zero ? MAR : PC)
    ctrl_start ctrl_load;                   // CNT <-- 0
…
end
```

Figure 8.153. Match between the contents of the memory that represents the microcode (the couple of memory banks **m1** and **m2** inside the control unit) and the microcode description itself.

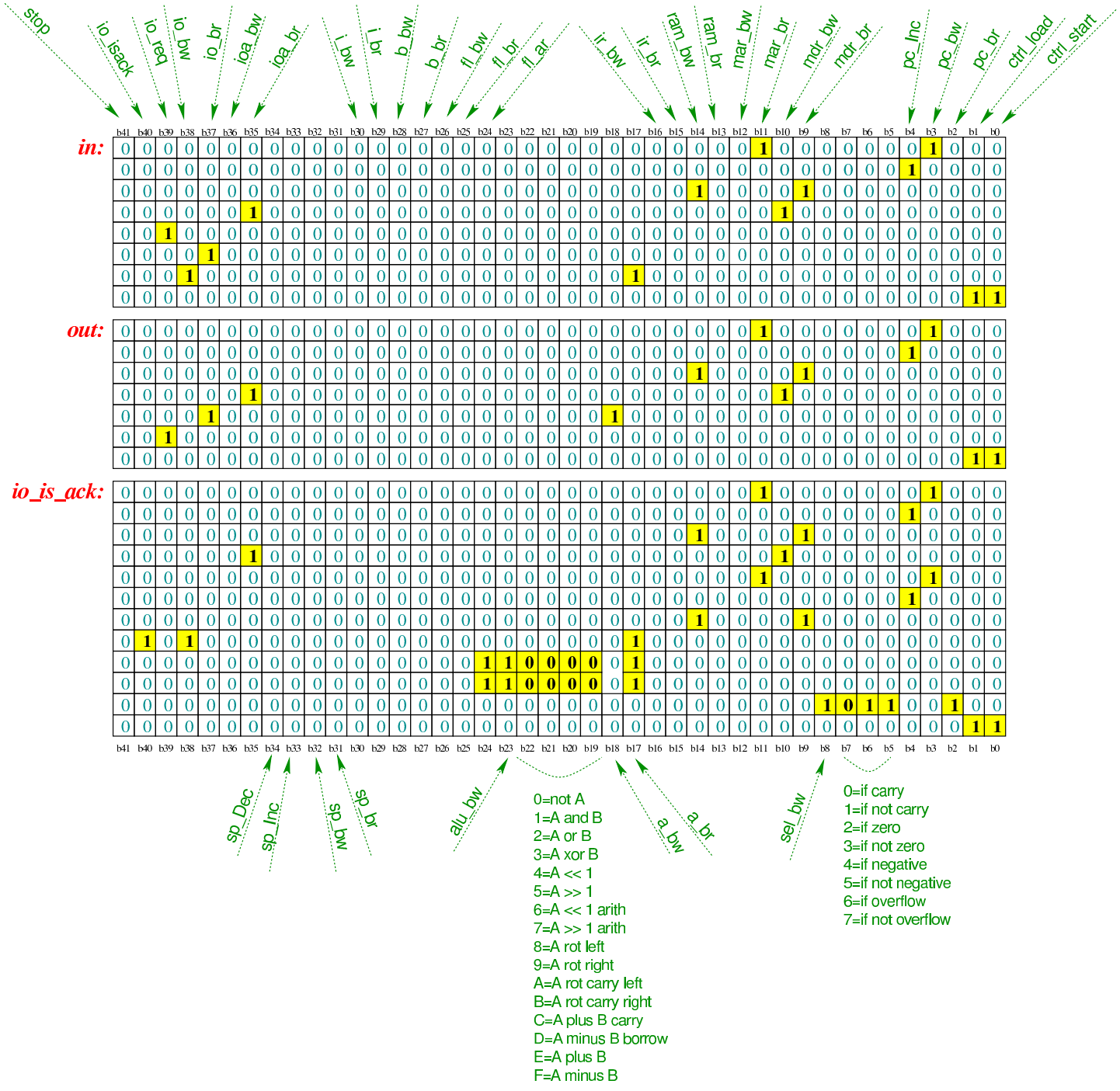


Table 8.154. Macroinstructions added for this version of the simple CPU.

Syntax	Description
`in` ***io_address***	It reads a byte from the I/O address specified by the argument.
`out` ***io_address***	It writes a byte to the I/O device specified by the argument.
`io_is_ack` ***io_address*** `#`***ref***	It reads a byte from the I/O device specified by the first argument. If the byte read is FF_{16} that means acknowledge ad it jumps to the address specified as the second argument.

8.8.7 Instruction "out"

Listing 8.155. Macrocode to experiment the use of **`out`** and **`io_is_ack`**. The program just shows the letter 'H' in a loop, without stop. The complete source file should be available at *attachments/xcpu/xcpu-a8-out.gm* .

```
begin macrocode @ 0
start:
        load_ref #character
        move_mdr_a
        out 0                       // display
check_ack:
        io_is_ack 0, #start
        jump #check_ack
stop:
        stop
character:
        .byte 0x48                  // 'H'
end
```

Video: ogv *http://www.youtube.com/watch?v=S9XqmTMYAj4* .

8.8.8 Instruction "in"

«

Listing 8.156. Macrocode to experiment the instruction **in**: the keyboard is read and the character read is output again with the **out** instruction, though the display. The complete source file should be available at *attachments/xcpu/xcpu-a8-in.gm* .

```
begin macrocode @ 0
start:
        in 1
        not
        not
        jump_if_zero #start
        out 0
        jump #start
stop:
        stop
end
```

Video: ogv *http://www.youtube.com/watch?v=JhGoQhssWQM* .

Version "B": optimization

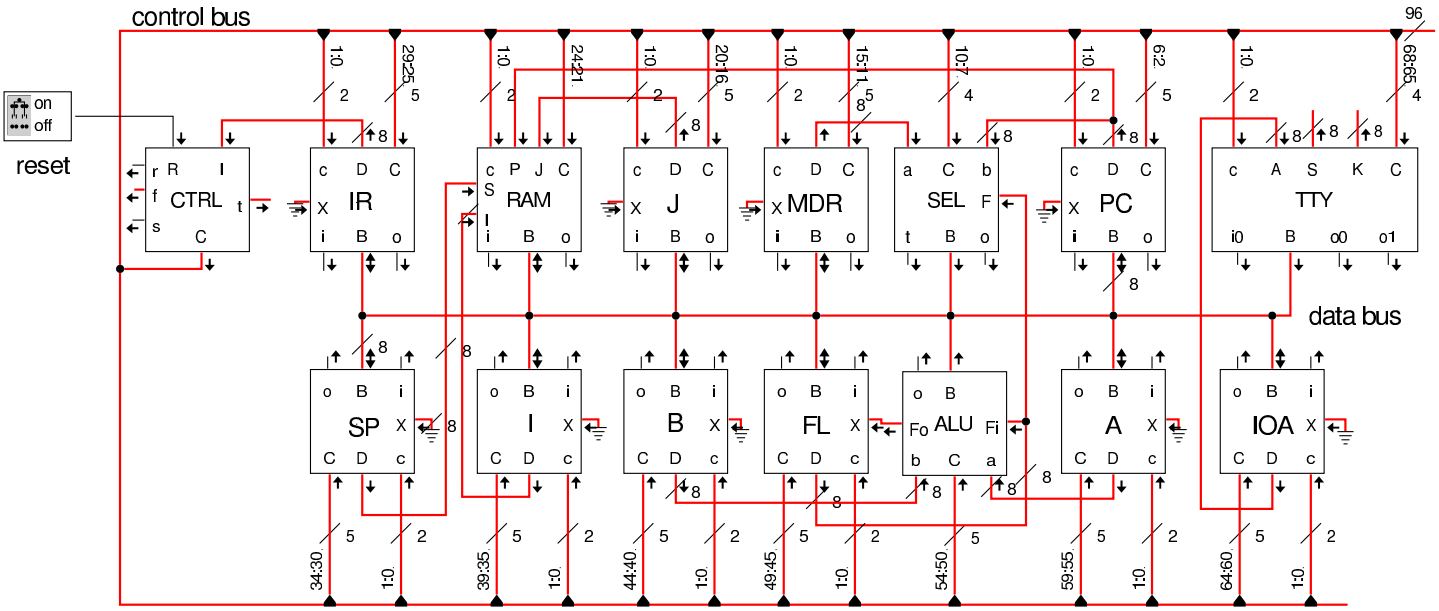
9.1 Uniform registers 446
9.2 Module "RAM" 449
9.3 Module "SEL" 453
9.4 Module "ALU" 453
9.5 Module "TTY": the terminal 454
9.6 Module "CTRL": control unit 459
9.7 Memories, fields, arguments and opcodes 463
9.8 Microcode 479
9.9 Macrocode: routine call 496
9.10 Macrocode: keyboard input and screen output 497
9.11 Version "B2" 498

Here is proposed a restructuring of the simple CPU developed up to this point, to rearrange and simplify its operation. Version "B1" starts from the implementation of uniform registers, up to an optimization of the operation, avoiding unnecessary clock cycles.

Attachments	Description
attachments/xcpu/xcpu-b1.v	TKGate Verilog netlist source file.
attachments/xcpu/xcpu-b1.gm	TKGate microcode and macrocode source file.
attachments/xcpu/xcpu-b1-terminal.vpd.tcl	TCL/Tk script related to the terminal module **TTY**.
attachments/xcpu/xcpu-b2.v	TKGate Verilog netlist source file.
attachments/xcpu/xcpu-b2.gm	TKGate microcode and macrocode source file.
attachments/xcpu/xcpu-b2-terminal.vpd.tcl	TCL/Tk script related to the terminal module **TTY**.

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

Figure 9.2. Simple CPU version "B1".



9.1 Uniform registers

All the registers of the new simple CPU have the opportunity to increase or reduce the value that they contain, by one unit; moreover, they have the ability to read a datum from the bus (***B***) or from an auxiliary input (***X***). In order to monitor their activity, these registers have two outputs where LEDs could be connected: these outputs are activated when reading or writing is requested.

Figure 9.3. Generalized registers, external appearance.

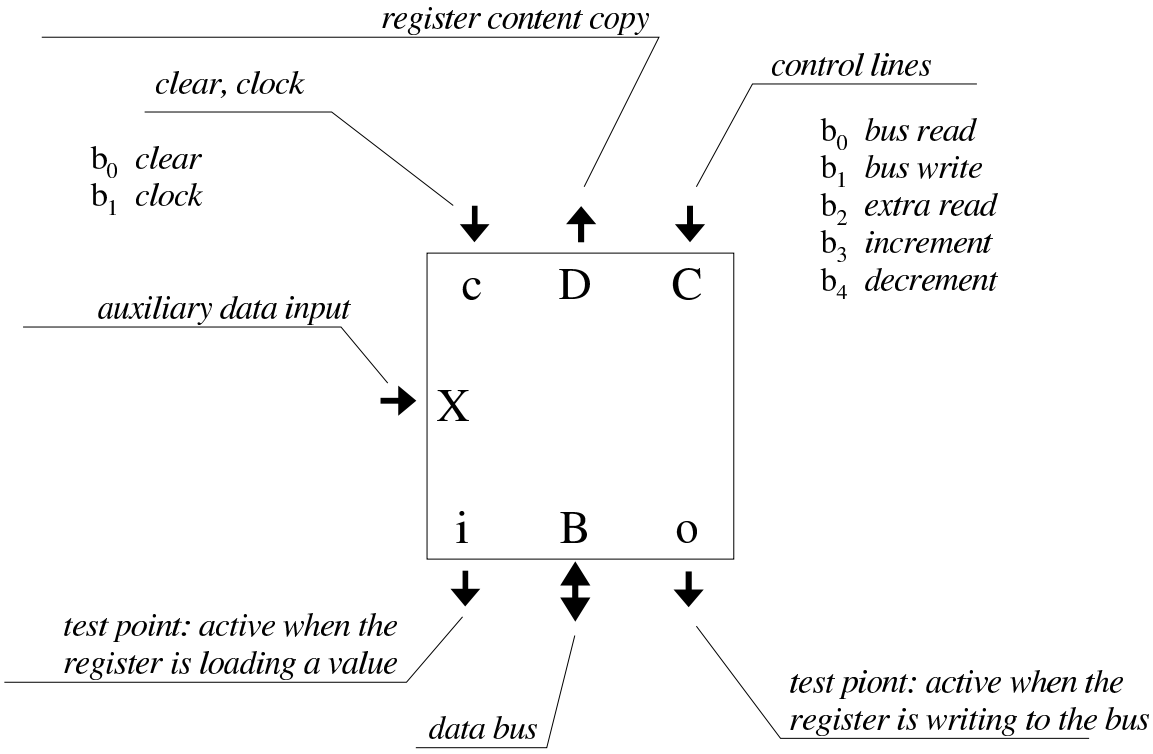
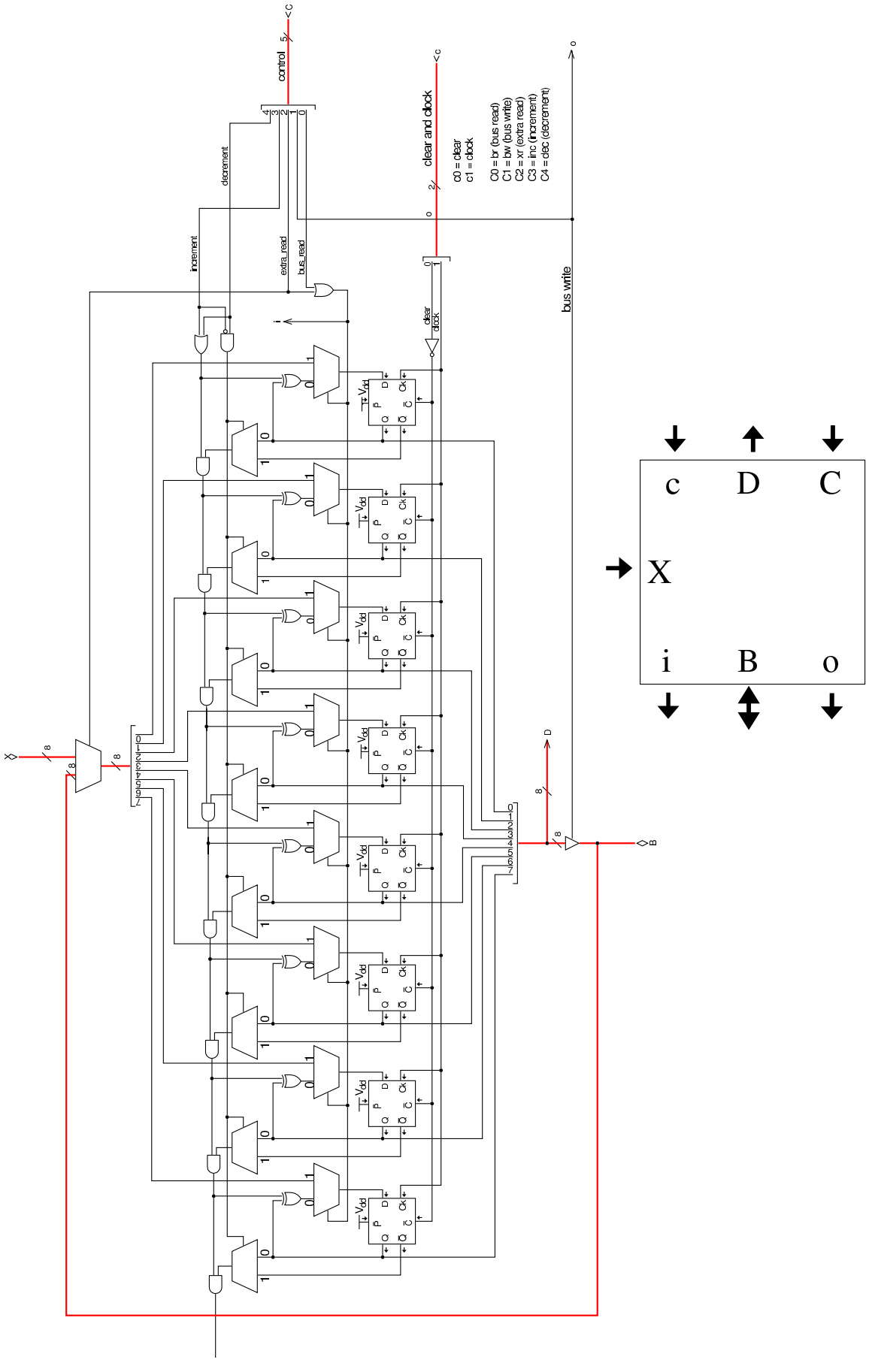


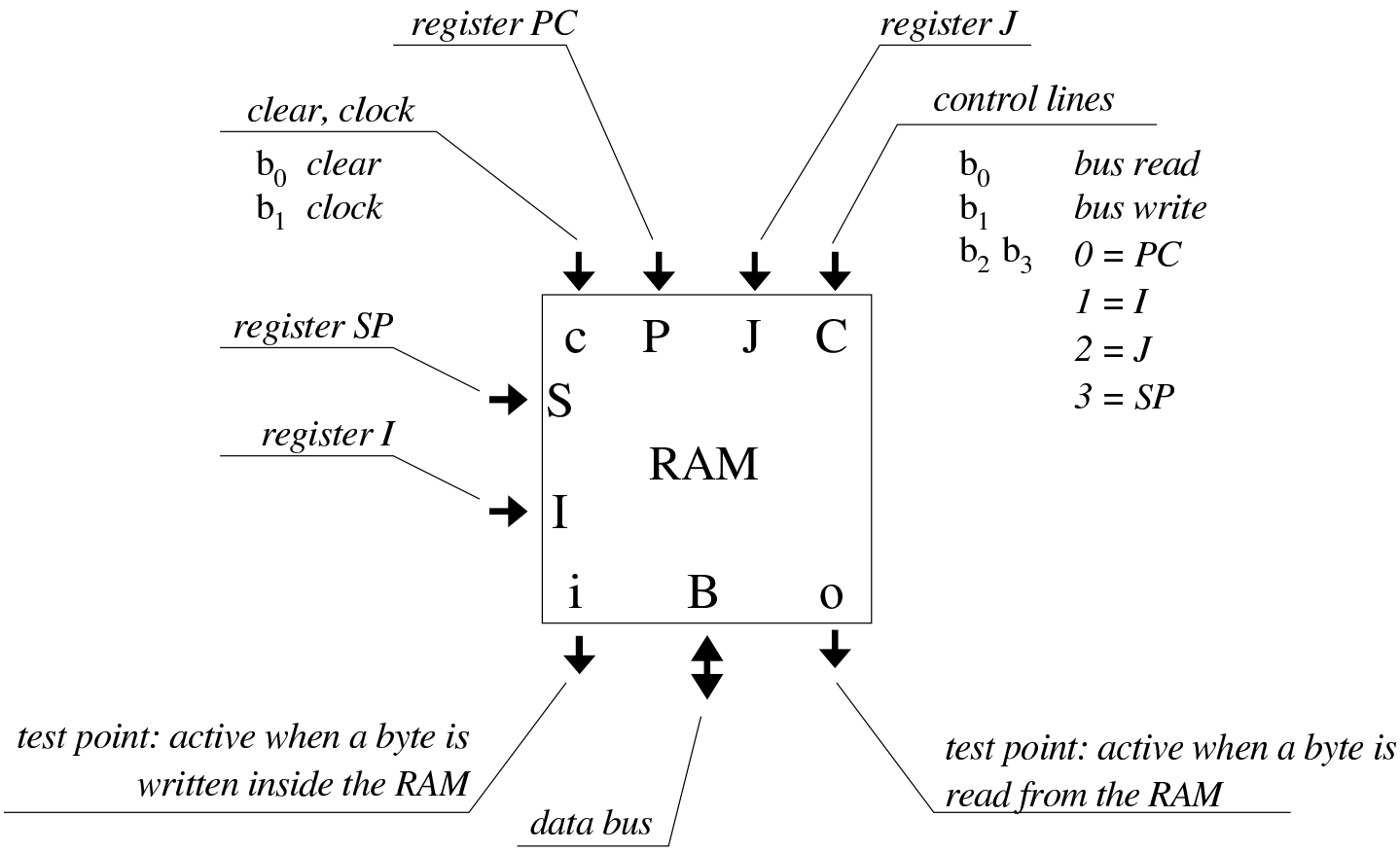
Figure 9.4. Generalized register: internal structure.



9.2 Module "RAM"

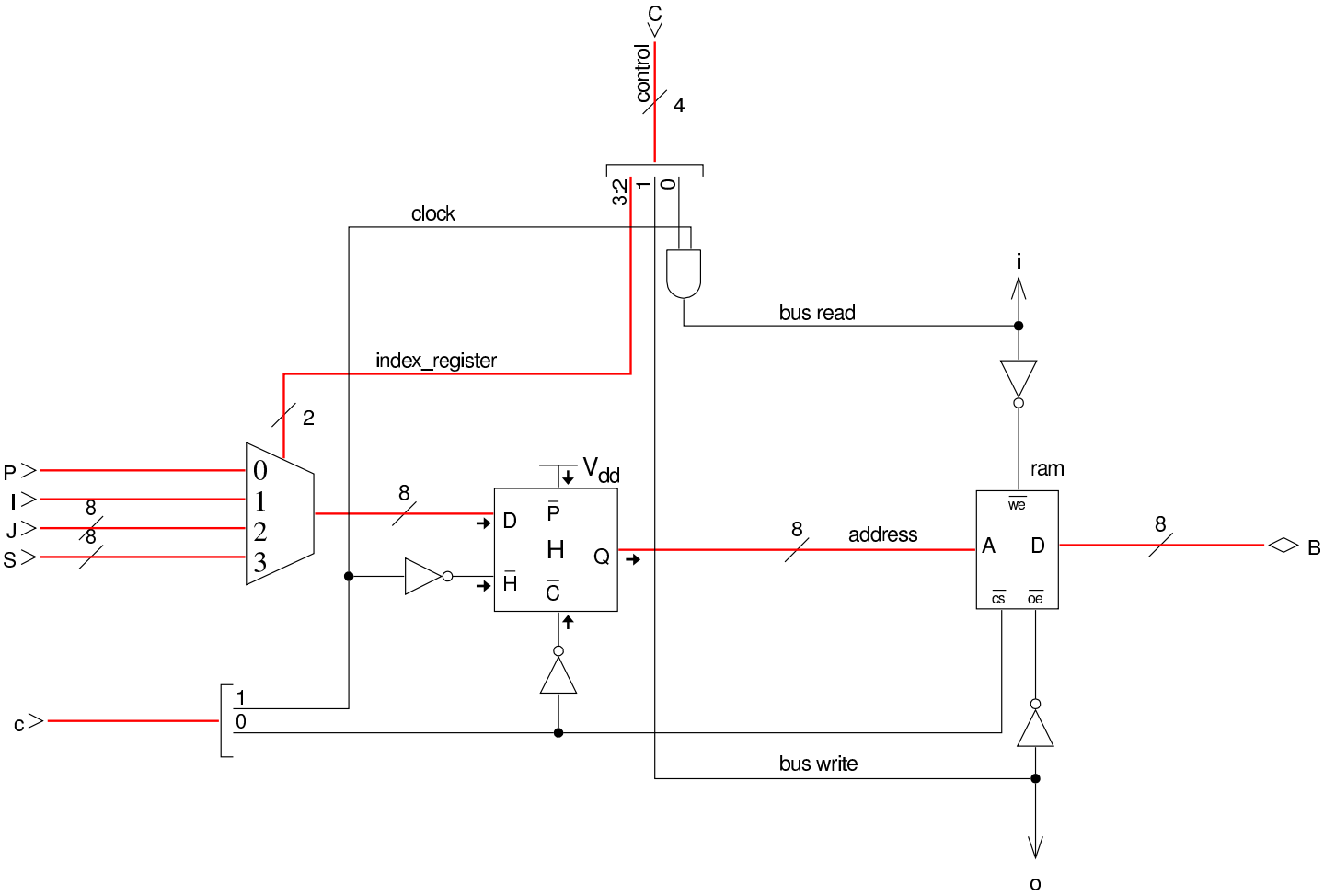
The **RAM** module may receive the address directly from the registers ***PC***, ***SP***, ***I*** and ***J***, without mediation, so the ***MAR*** used until the earlier version is now removed. The choice of the address register to read depends on the code contained in the group of lines of the ***C*** input.

Figure 9.5. Module **RAM**, external appearance.



The internal diagram of the **RAM** module changes substantially, allowing to use the address from the selected register, but only at the state in which this data is valid. In the diagram there is the addition of a module, called **H**, which is a register controlled by an enable input. Therefore, this register does not react to the change of the clock pulse, it is used to maintain stored a value for all the time in which the input $\overline{H}$ is asserted (zero, as it is a complemented input line).

Figure 9.6. Module **RAM**, internal structure.

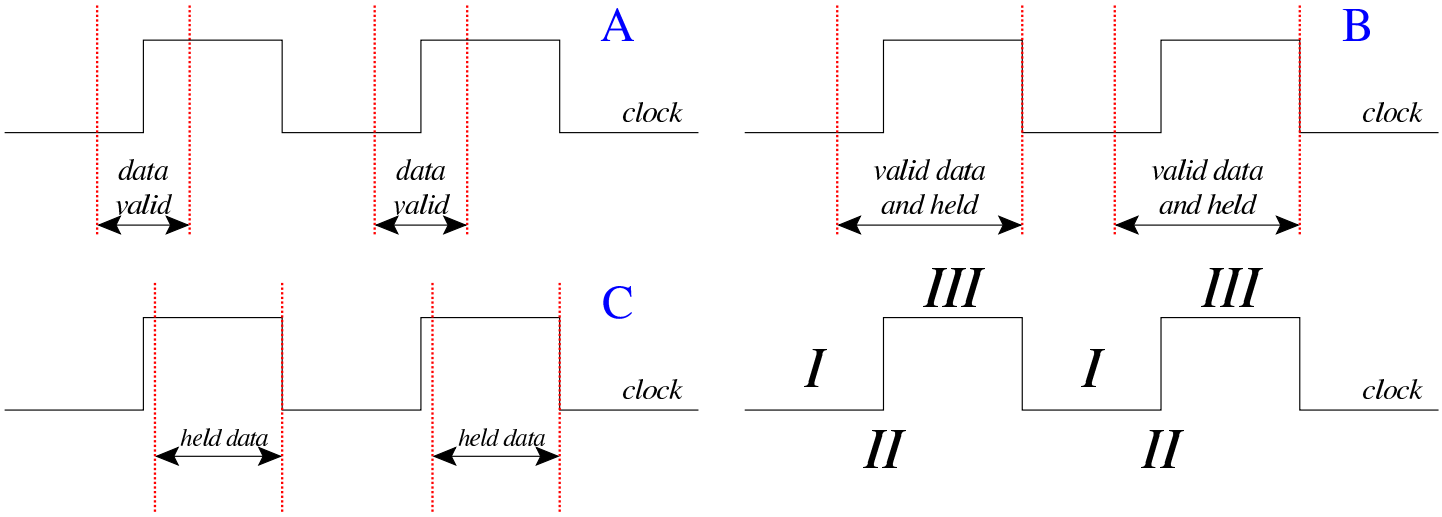


In this version of the CPU, during a clock cycle, the address used to locate a memory cell may not be stable, due to various factors. First of all, the address is selected, by a multiplexer, from four different registers, but this selection can be done at the start of the 'I' phase shown inside the next figure; therefore, in this first phase the address received by the **RAM** module changes in the majority of cases. Then, when the clock signal changes from zero to one, the address source register might be updated in preparation for the next step. Therefore, the valid information from the address used for the RAM appears just before the clock positive edge (phase 'II'). However, when it is required to write a value in the RAM, the RAM itself needs to have the address for a certain time, during which this address should not change: this is done with the **H** module that is transparent when the signal clock is zero, while it blocks its value when the clock signal is active. For that reason, the RAM is enabled for receiving a request to read or write only during the active period of the clock signal (phase 'III'). When the RAM is just read, it is sufficient that the RAM has had time to provide the data corresponding to the address selected when the information is then loaded from

the bus by another register.

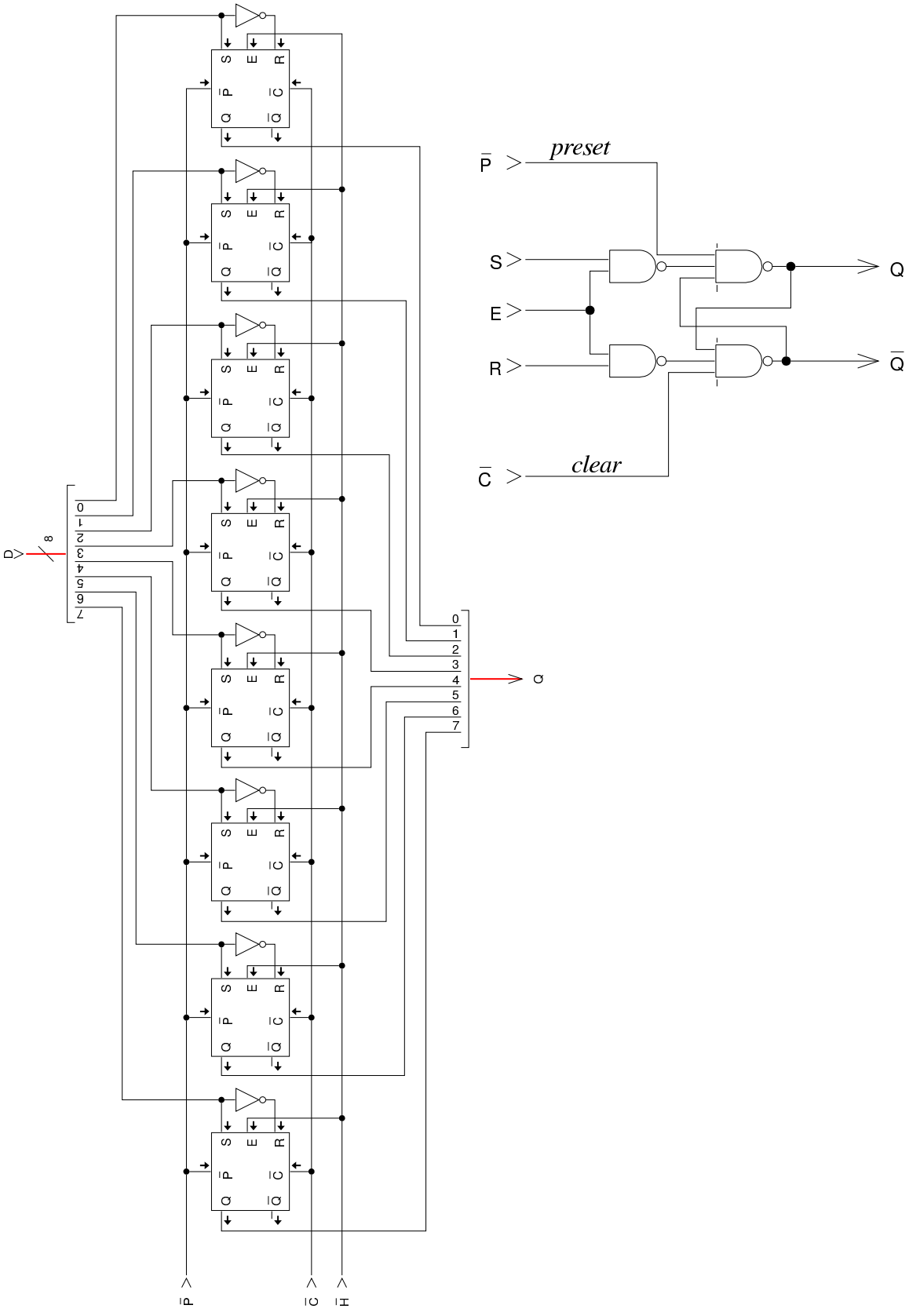
Inside the figure, the graph 'A' refers to the ranges of validity of the address information, around the positive variation of the clock signal. The graph 'B' shows the situation at the output of module **H**, which extends the validity of the incoming address, because when the clock signal becomes positive, the module **H** blocks the value at its output. The graph 'C' shows the period in which it is granted to the RAM memory to change its content (RAM write).

Figure 9.7. Functional phases for the **RAM** module.



The module **H** is made of simple SR flip-flops, connected so as to operate as D flip-flops, with the enable input. Simple flip-flops with enable input are used to avoid the introducing of excessive latencies.

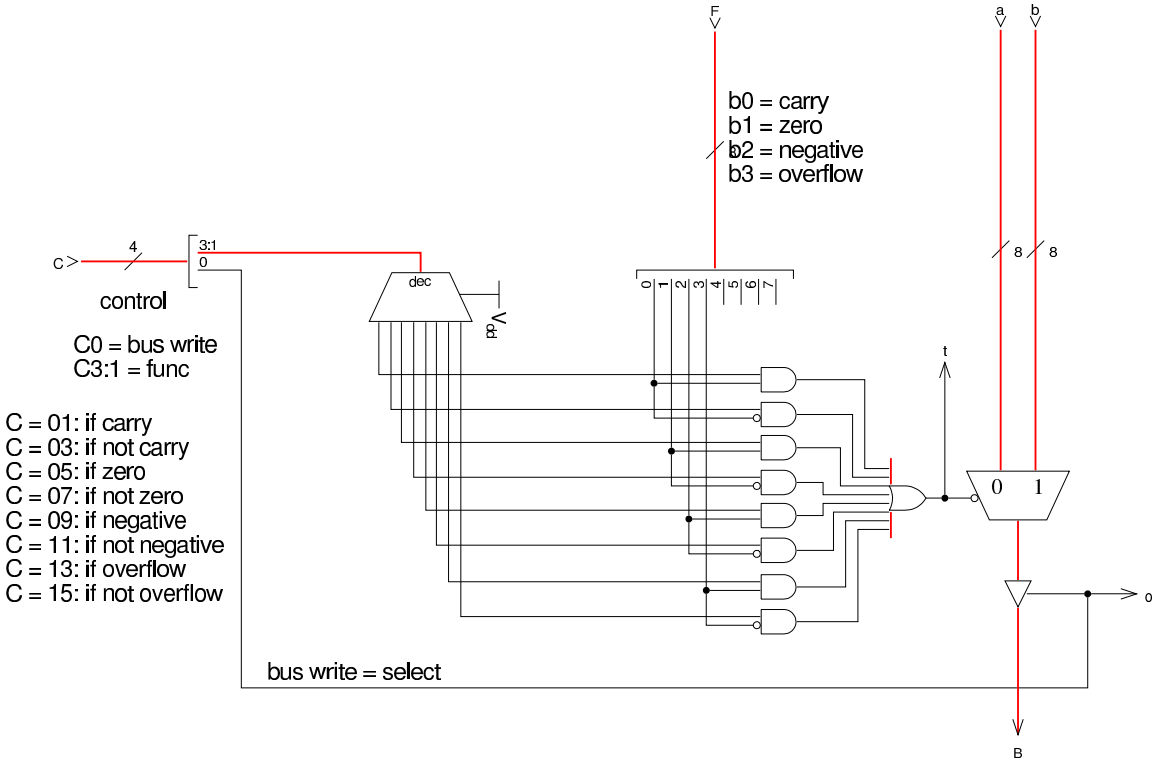
Figure 9.8. Module **H** (*hold*), internal structure.



9.3 Module "SEL"

The selection module is not changed, except for the reorganization of the wiring and the addition of a diagnostic output to know when the evaluated condition is fulfilled (output ***t***).

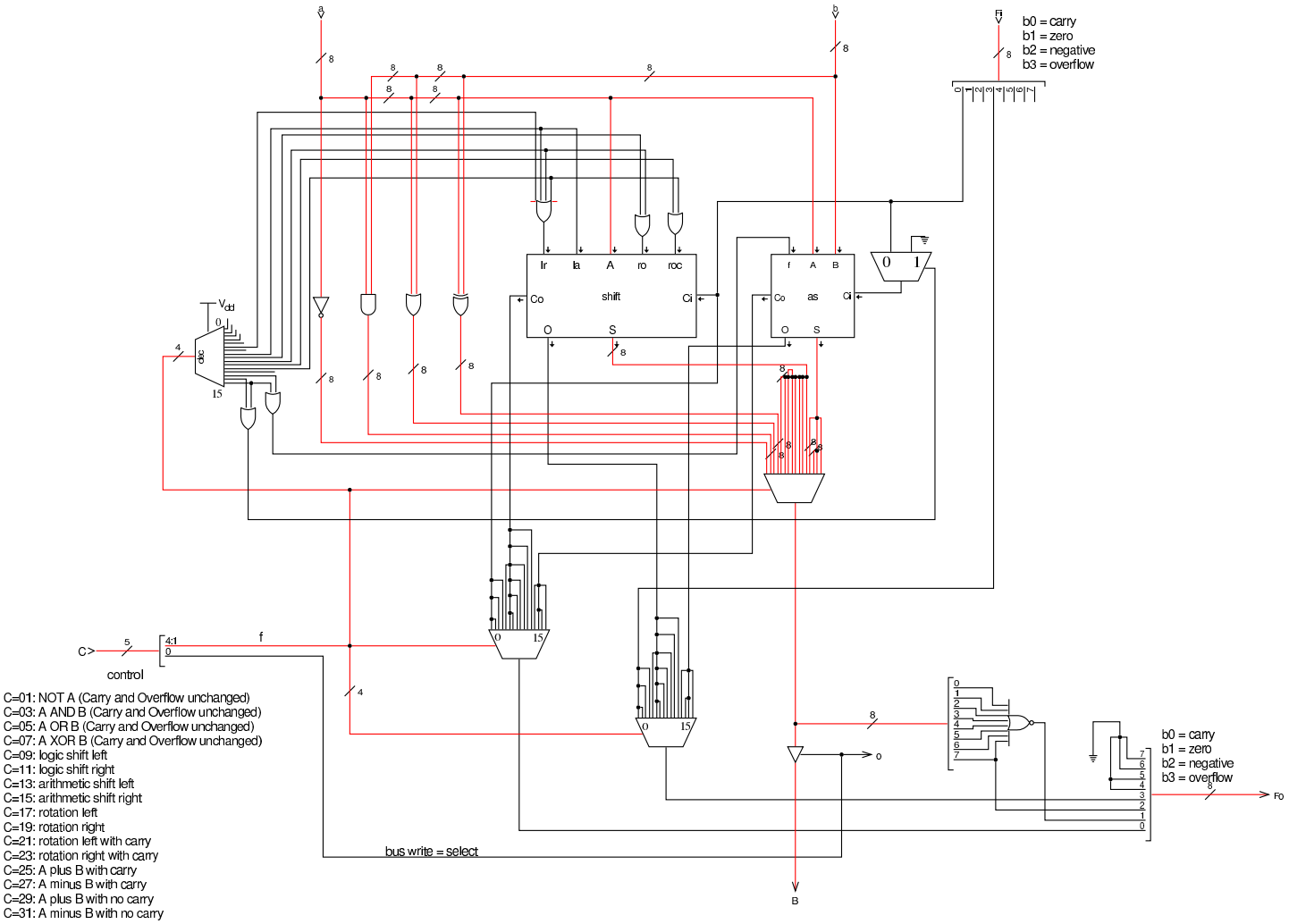
Figure 9.9. Module **SEL**, internal structure.



9.4 Module "ALU"

Also the **ALU** module is not substantially changed from the previous one: the control lines are joined together and there is a new diagnostic indicator (output ***o***) that is active when the module writes to the bus.

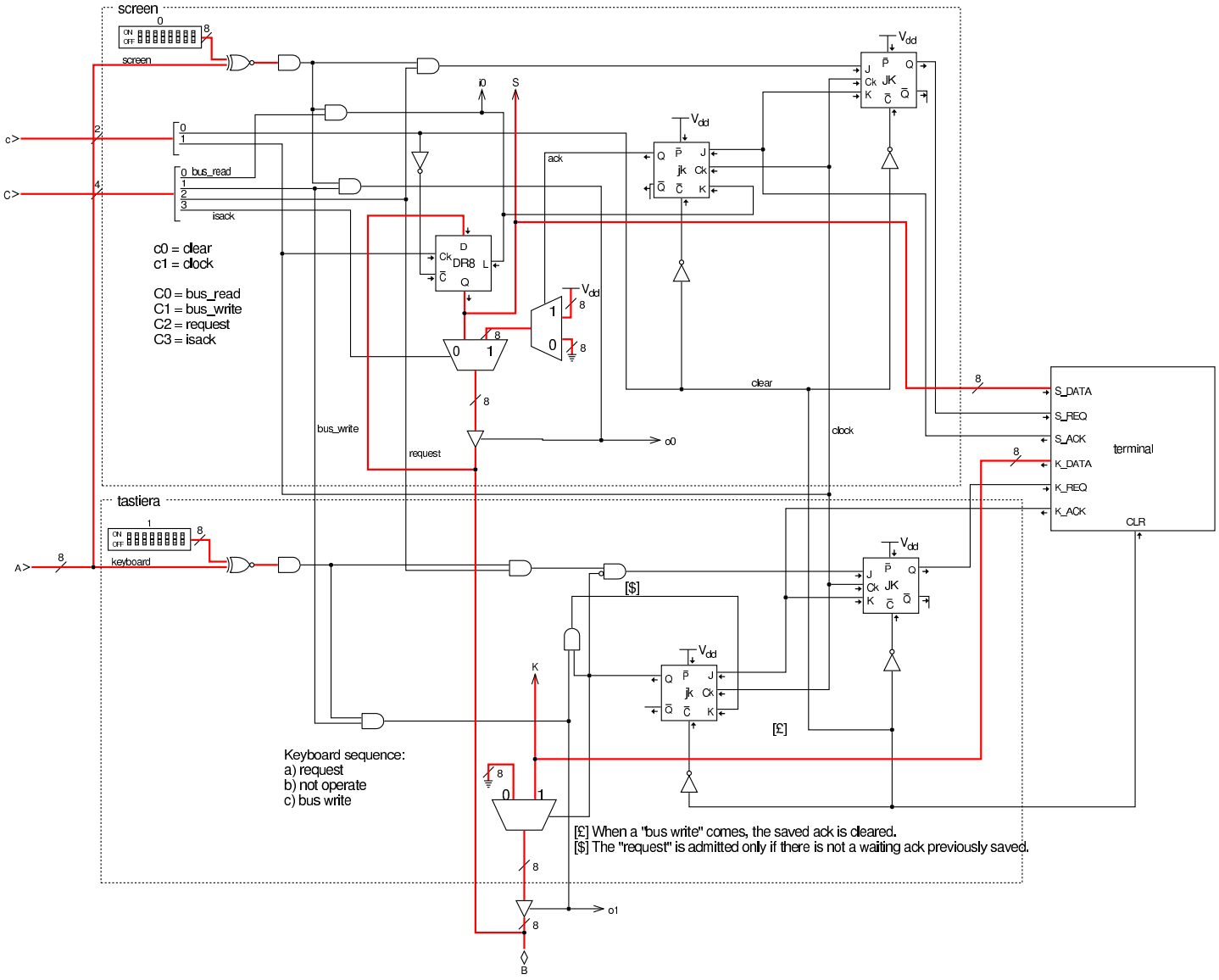
Figure 9.10. Module **ALU**, internal structure.



9.5 Module "TTY": the terminal

The terminal, made of the keyboard-screen group, is different from the previous version of the simple CPU, because it is unified, as it is implemented already for the TKGate library. However, the unification internally maintains the circuit distinction of the previous version and also the same operating logic; in practice, keyboard and screen continue to be managed separately, but the TCL/Tk implementation code is made of a single module, which manifests itself in a single window during the TKGate simulation.

Figure 9.11. Module **TTY**, internal structure.



In the figure that shows the circuit of the **TTY** module there is a delimitation between the two portions, related to keyboard and screen: it should be noted that the two blocks are activated via different addresses (input ***A***), exactly as it is in the previous version. The **terminal** module is written in Verilog, as already done in the previous version, but in this case it is a single module, for keyboard and screen. In turn, the **terminal** module uses TCL/Tk code that is contained by the file '`terminal.tcl`' shown immediately after.

Listing 9.12. Module **terminal**, written in Verilog.

```
module terminal(K_DATA, K_REQ, K_ACK, S_DATA, S_REQ, S_ACK, CLR);

output K_ACK;
output S_ACK;
output [7:0] K_DATA;
input [7:0] S_DATA;
input K_REQ;
input S_REQ;
input CLR;
reg k_ready;
reg [7:0] key;
reg s_ready;

  initial
    begin
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  always
    begin
      @(posedge CLR)
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  initial $tkg$post("TERMINAL","%m");

  always
    begin
      @ (posedge K_REQ);
      # 5;
      key = $tkg$recv("%m.KD");
      # 5;
```

```
      k_ready = 1'b1;
      # 5;
      @ (negedge K_REQ);
      # 5;
      k_ready = 1'b0;
    end

  always
    begin
      @(posedge S_REQ);
      # 5;
      $tkg$send("%m.SD",S_DATA);
      # 5;
      s_ready = 1'b1;
      # 5;
      @(negedge S_REQ);
      # 5;
      s_ready = 1'b0;
    end

  assign S_ACK = s_ready;
  assign K_DATA = key;
  assign K_ACK = k_ready;

endmodule
```

Listing 9.13. File ‘`share/tkgate/vpd/terminal.tcl`’. The file is similar to the one included in the TKGate library, for the terminal management.

```
image create bitmap txtcurs -file "$bd/txtcurs.b"

VPD::register TERMINAL
VPD::allow TERMINAL::post
VPD::allow TERMINAL::data

namespace eval TERMINAL {
    # Public variables declarations: the variables $terminal_...
    # are arrays of which only the element $n is used;
    # that element identifies uniquely the working interface instance.
    variable terminal_w
    variable terminal_pos
```

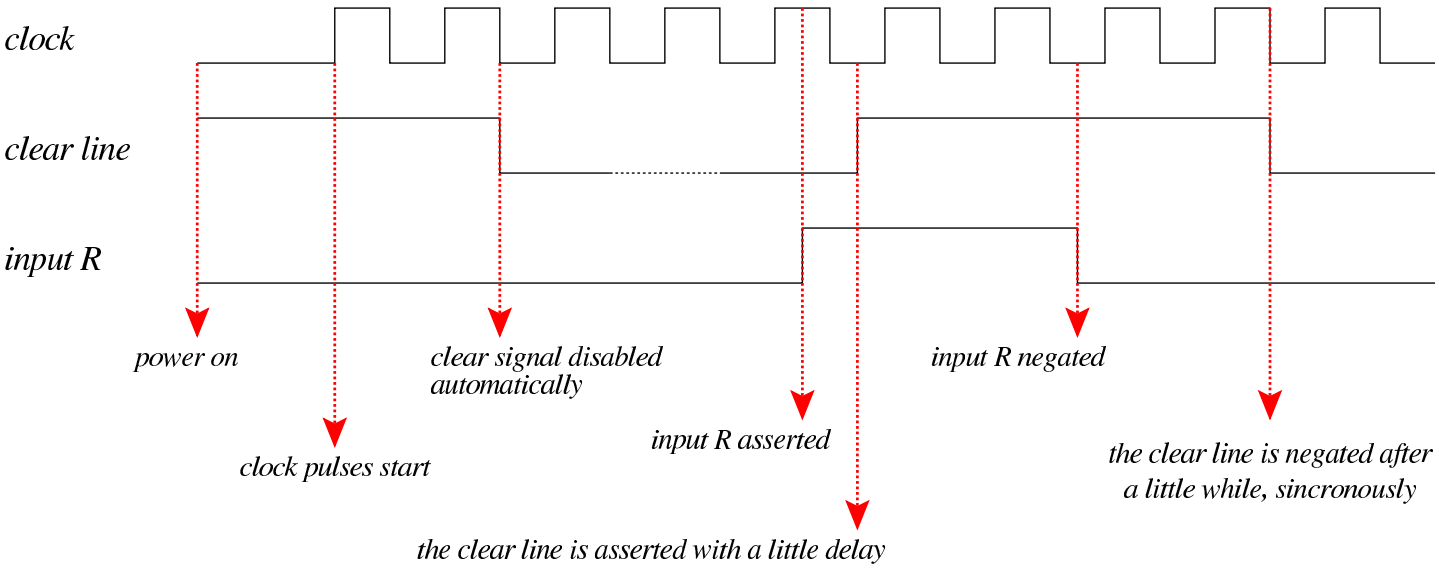
```
    #
    variable KD
    # Function requested by TKGate to create the interface.
    proc post {n} {
        variable terminal_w
        variable terminal_pos
        # Create the window and save the object element in a $terminal_w array element.
        set terminal_w($n) [VPD::createWindow "TERMINAL $n" -shutdowncommand "TERMINAL::unpost $n"]
        # For convenience, copy the object reference inside the local
        # variable $w; then, the variable $w will be used as a reference to the object.
        set w $terminal_w($n)
        text $w.txt -state disabled
        pack $w.txt
        # Put the cursor at the end of the displayed text.
        $w.txt image create end -image txtcurs
        # Bind the keyboard input, related to the object represented by
        # $terminal_w($n), to the function sendChar.
        bind $w <KeyPress> "TERMINAL::sendChar $n \"%A\""
        # Open a reading channel, named «SD» (screen data),
        # and associate it to the function «data»; moreover, open a
        # writing channel, named «KD» (keyboard data).
        if {[info exists ::tkgate_isInitialized]} {
            VPD::outsignal $n.KD TERMINAL::KD($n)
            VPD::insignal $n.SD -command "TERMINAL::data $n" -format %d
        }
        # Reset the character count, used to count the characters displayed
        # on screen.
        set terminal_pos($n) 0
    }
    # Function that receives the typing and put it into the
    # channel «KD», related to the current interface instance.
    proc sendChar {n key} {
        variable KD
        if { [string length $key ] == 1 } {
            binary scan $key c c
            set TERMINAL::KD($n) $c
        }
    }
    # Function that TKGate requires to destroy the interface.
    proc unpost {n} {
        variable terminal_w
        variable terminal_pos
        destroy $terminal_w($n)
        destroy $terminal_pos($n)
        unset terminal_w($n)
        unset terminal_pos($n)
    }
    # Function used to get the data to display on screen.
    proc data {n c} {
        variable terminal_w
        variable terminal_pos
        # For convenience, copy the object reference representing the
```

```
        # interface, inside the variable $w.
        set w $terminal_w($n)
        catch {
            # The variable $c contains the character to display.
            if { $c == 7 } {
                # BEL
                bell
                return
            } elseif { $c == 127 || $c == 8 } {
                # DEL | BS
                if { $terminal_pos($n) > 0 } {
                    # Delete the last displayed character, but only if the
                    # characters counter is greater than zero, otherwise
                    # the cursor would disappear and the next characters
                    # would be located in an unvisible screen area.
                    $w.txt configure -state normal
                    $w.txt delete "end - 3 chars"
                    $w.txt see end
                    $w.txt configure -state disabled
                    set terminal_pos($n) [expr {$terminal_pos($n) - 1}]
                }
                return
            } elseif { $c == 13 } {
                # Convert CR to LF.
                set c 10
            }
            # Convert the character number into a visible symbol.
            set x [format %c $c]
            # Display the symbol.
            $w.txt configure -state normal
            $w.txt insert "end - 2 chars" $x
            $w.txt see end
            $w.txt configure -state disabled
            # Update the displayed characters counter.
            set terminal_pos($n) [expr {$terminal_pos($n) + 1}]
        }
    }
}
```

9.6 Module "CTRL": control unit

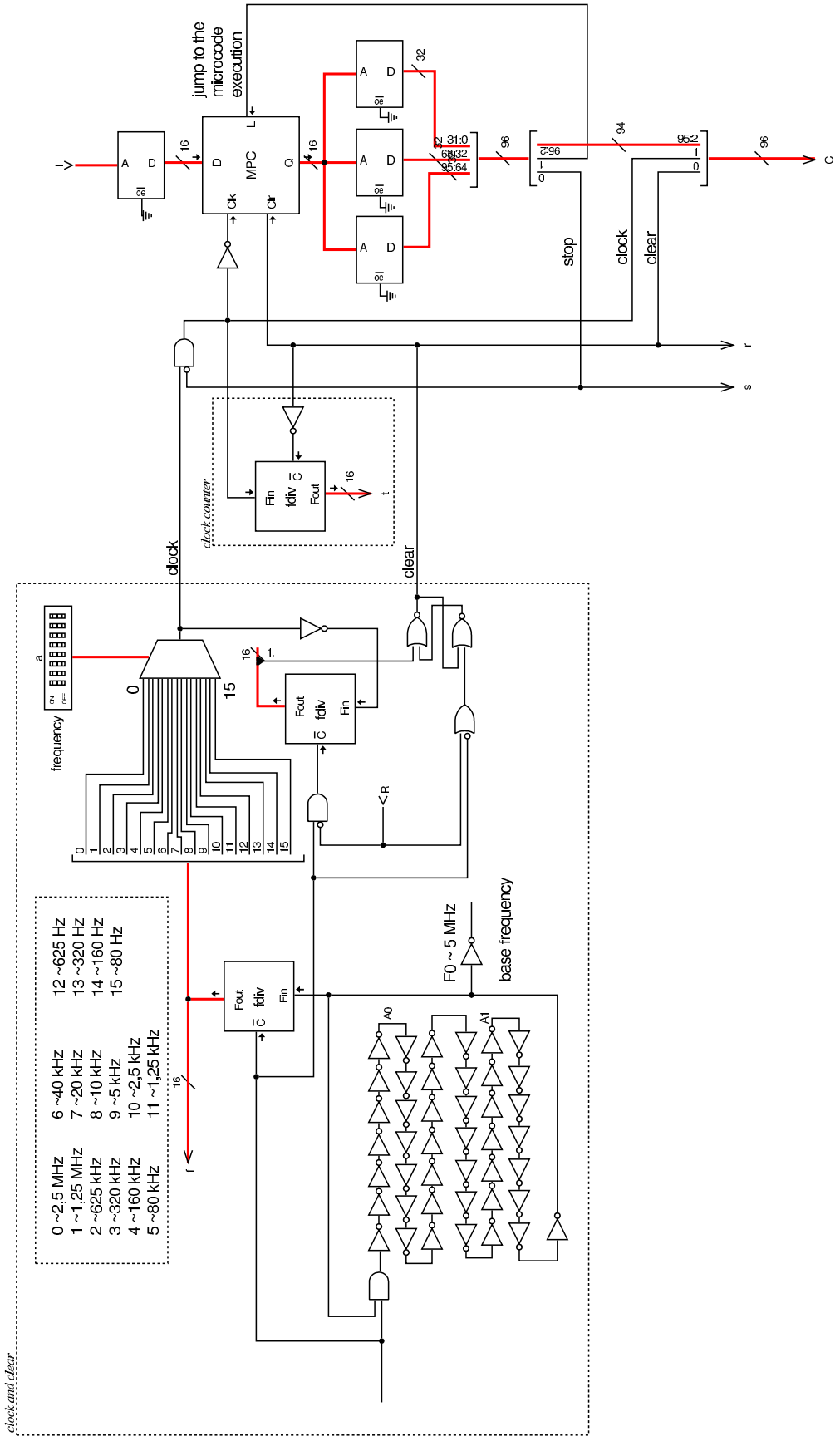
To simplify the organization of the wiring, the control unit incorporates the generator of clock pulses; moreover, the generator of clock pulses incorporates the management of the reset signal, so that it is removed only at the moment more suitable for the impulse clock: up to the previous version of the simple CPU, the circuit required a manual reset before the CPU could work properly, but the release of the reset signal could happen at an unsuitable time, which made it all unstable.

Figure 9.14. Timing for the ***clear*** line.



The following picture shows the block diagram of the control unit that integrates the clock functionality. In the left part is the circuit that generates the clock pulses and controls the ***clear*** line. The module **`fdiv`** is extended compared to the previous version, in order to be able to divide more the frequency. The frequency selection is made through a multiple switch (dip switch) connected to a multiplexer that is seen at the top. However, the experiments with TKGate show that the CPU can operates with a maximum clock frequency of 1.25 MHz that is produced setting this switch to the value 1.

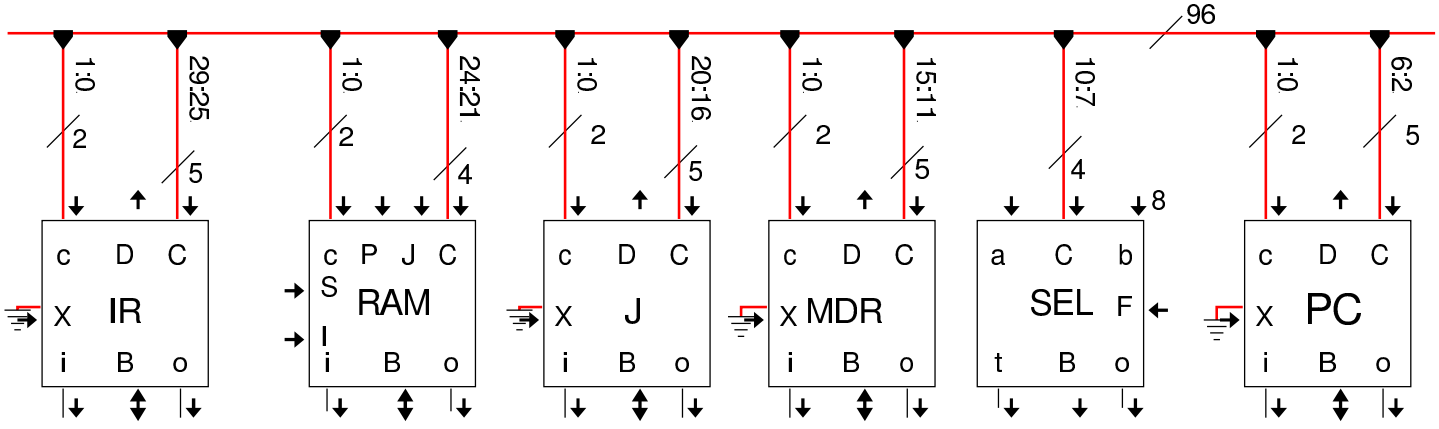
Figure 9.15. Control unit with the clock generator.



There are different **fdiv** modules inside the control unit: the central one is used to count the pulses to synchronize the reset line, the rightmost is used to count clock pulses from the start and enable monitoring through the ***t*** output: it is therefore only of a diagnostic aid.

On the right part of the figure that represents the original control unit, there is a unique 16-bit counter (**MCP**) and it lacks the opportunity to restore the execution of the microcode from the beginning (address zero). The two least significant bit of the control word are used to bring the clock pulse and the reset signal (clear); the same two least significant lines that come out from the memory that contains the microcode, are used to control the module itself and do not affect the rest of the CPU. This version of the control unit does not allow to resume the clock signal after the internal ***stop*** line is asserted.

Figure 9.16. Connections to the control bus.



On the picture above there are some components of the simple CPU, connected to the control bus. All these components have in common the ***c*** input (lowercase) and the ***C*** input (upper case). The ***c*** input is always connected to the first two lines of the bus control, from which it is obtained, respectively, the reset signal and the clock signal. The ***C*** input, instead, must be connected to the lines of the control bus that specifically relate to the module. In the case of generalized registers, these lines are always five: ***bus_read***, ***bus_write***, ***extra_read***, ***increment***, ***decrement***. The ***PC*** register connects its ***C*** input to the lines 2 to 6 (6:2) of the control bus; the module **SEL** (which uses only four control lines) is connected to the lines 7 to 10 (10:7), and so is with the other components.

9.7 Memories, fields, arguments and opcodes

The TKGate source used to describe the contents of the memory used by the simple CPU, always begins with the definition of the memory dimensions, along with each bank name:

```
map       bank[7:0]   ctrl.map;
microcode bank[31:0]  ctrl.micro0;
microcode bank[63:32] ctrl.micro1;
microcode bank[91:64] ctrl.micro2;
macrocode bank[7:0]   ram.ram;
```

In this version of the simple CPU there are some changes to the memory names, in order to make clearer each respective task. It should be noted that three memory modules, each of 32 bits, are used for the microcode word, because the control bus requires the use of many lines.

```
field ctrl[1:0]  = {nop=0, stop=1, load=2};
field pc[6:2]    = {br=1, bw=2, xr=4, inc=8, dec=16};
field sel[10:7]  = {if_carry=1, if_not_carry=3,
                    if_zero=5, if_not_zero=7,
                    if_negative=9, if_not_negative=11,
                    if_overflow=13, if_not_overflow=15};
field mdr[15:11] = {br=1, bw=2, xr=4, inc=8, dec=16};
field j[20:16]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field ram[24:21] = {br=1, bw=2, p=0, i=4, j=8, s=12};
field ir[29:25]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field sp[34:30]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field i[39:35]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field b[44:40]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field fl[49:45]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field alu[54:50] = {not=1, and=3, or=5, xor=7, lshl=9, lshr=11,
                    ashl=13, ashr=15, rotl=17, rotr=19, rotcl=21,
                    rotcr=23, add_c=25, sub_b=27, add=29, sub=31};
field a[59:55]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field ioa[64:60] = {br=1, bw=2, xr=4, inc=8, dec=16};
field ioc[68:65] = {br=1, bw=2, req=4, isack=8};
```

The control word fields are written in a more compact manner. It should be noted

that values that can be represented in each field may be added with the binary OR operator. In practice, in relation to the field ***pc***, which refers to the ***PC*** register control lines, both the writing to the data bus (**pc=bw**) and the increment (**pc=inc**) could be turned on, during the same clock cycle.

Inside the **ctrl** field declaration there are the first two lines, used for the stop clock and microinstruction load, but it is worth to remaind that those two lines are not connected to the control bus outside, because in their place there are the clear and clock lines.

In this version of the project it is possible to declare a null microinstruction, with the sole purpose of waiting a clock cycle. This microinstruction is selected with: **ctrl=nop**.

The opcode remains 8-bit long, then, there may be a maximum of two arguments (8 bits each):

```
operands op_0 {
    -  = { };
};
operands op_1 {
    #1 = { +1=#1[7:0]; };
};
operands op_2 {
    #1,#2 = { +1=#1[7:0]; +2=#2[15:8]; };
};
```

The opcode of the available macroinstructions is simply an integer that starts from zero, matching the **nop** instruction, ending with 255 which matches the **stop** instruction. There are no other considerations.

```
op nop {                    // not operate
  map nop: 0;
  +0[7:0]=0;
  operands op_0;
};
op load                     // MDR <-- RAM[arg]
{
  map load: 1;
```

```
  +0[7:0]=1;
  operands op_1;
};
op load_i                    // MDR <-- RAM[i]
{
  map load_i: 2;
  +0[7:0]=2;
  operands op_0;
};
op load_j                    // MDR <-- RAM[j]
{
  map load_j: 3;
  +0[7:0]=3;
  operands op_0;
};
op store {                   // RAM[arg] <-- MDR
  map store: 4;
  +0[7:0]=4;
  operands op_1;
};
op store_i {                 // RAM[i] <-- MDR
  map store_i: 5;
  +0[7:0]=5;
  operands op_0;
};
op store_j {                 // RAM[j] <-- MDR
  map store_j: 6;
  +0[7:0]=6;
  operands op_0;
};
op cp_ij {                   // RAM[j++] <-- MDR <-- RAM[i++]
  map cp_ij: 7;
  +0[7:0]=7;
  operands op_0;
```

```
};
op cp_ji {                  // RAM[i++] <-- MDR <-- RAM[j++]
  map cp_ji: 8;
  +0[7:0]=8;
  operands op_0;
};
op mv_mdr_a {               // A <-- MDR
  map mv_mdr_a: 9;
  +0[7:0]=9;
  operands op_0;
};
op mv_mdr_b {               // B <-- MDR
  map mv_mdr_b: 10;
  +0[7:0]=10;
  operands op_0;
};
op mv_mdr_fl {              // FL <-- MDR
  map mv_mdr_fl: 11;
  +0[7:0]=11;
  operands op_0;
};
op mv_mdr_sp {              // SP <-- MDR
  map mv_mdr_sp: 12;
  +0[7:0]=12;
  operands op_0;
};
op mv_mdr_i {               // I <-- MDR
  map mv_mdr_i: 13;
  +0[7:0]=13;
  operands op_0;
};
op mv_mdr_j {               // J <-- MDR
  map mv_mdr_j: 14;
  +0[7:0]=14;
```

```
  operands op_0;
};
op mv_a_mdr {              // A <-- MDR
  map mv_a_mdr: 15;
  +0[7:0]=15;
  operands op_0;
};
op mv_a_b {                // B <-- A
  map mv_a_b: 16;
  +0[7:0]=16;
  operands op_0;
};
op mv_a_fl {               // FL <-- A
  map mv_a_fl: 17;
  +0[7:0]=17;
  operands op_0;
};
op mv_a_sp {               // SP <-- A
  map mv_a_sp: 18;
  +0[7:0]=18;
  operands op_0;
};
op mv_a_i {                // I <-- A
  map mv_a_i: 19;
  +0[7:0]=19;
  operands op_0;
};
op mv_a_j {                // J <-- A
  map mv_a_j: 20;
  +0[7:0]=20;
  operands op_0;
};
op mv_b_a {                // A <-- B
  map mv_b_a: 21;
```

```
  +0[7:0]=21;
  operands op_0;
};
op mv_b_mdr {              // MDR <-- B
  map mv_b_mdr: 22;
  +0[7:0]=22;
  operands op_0;
};
op mv_b_fl {               // FL <-- B
  map mv_b_fl: 23;
  +0[7:0]=23;
  operands op_0;
};
op mv_b_sp {               // SP <-- B
  map mv_b_sp: 24;
  +0[7:0]=24;
  operands op_0;
};
op mv_b_i {                // I <-- B
  map mv_b_i: 25;
  +0[7:0]=25;
  operands op_0;
};
op mv_b_j {                // J <-- B
  map mv_b_j: 26;
  +0[7:0]=26;
  operands op_0;
};
op mv_fl_a {               // A <-- FL
  map mv_fl_a: 27;
  +0[7:0]=27;
  operands op_0;
};
op mv_fl_b {               // B <-- FL
```

```
  map mv_fl_b: 28;
  +0[7:0]=28;
  operands op_0;
};
op mv_fl_mdr {          // MDR <-- FL
  map mv_fl_mdr: 29;
  +0[7:0]=29;
  operands op_0;
};
op mv_fl_sp {           // SP <-- FL
  map mv_fl_sp: 30;
  +0[7:0]=30;
  operands op_0;
};
op mv_fl_i {            // I <-- FL
  map mv_fl_i: 31;
  +0[7:0]=31;
  operands op_0;
};
op mv_fl_j {            // J <-- FL
  map mv_fl_j: 32;
  +0[7:0]=32;
  operands op_0;
};
op mv_sp_a {            // A <-- SP
  map mv_sp_a: 33;
  +0[7:0]=33;
  operands op_0;
};
op mv_sp_b {            // B <-- SP
  map mv_sp_b: 34;
  +0[7:0]=34;
  operands op_0;
};
```

```
op mv_sp_fl {                    // FL <-- SP
  map mv_sp_fl: 35;
  +0[7:0]=35;
  operands op_0;
};
op mv_sp_mdr {                   // MDR <-- SP
  map mv_sp_mdr: 36;
  +0[7:0]=36;
  operands op_0;
};
op mv_sp_i {                     // I <-- SP
  map mv_sp_i: 37;
  +0[7:0]=37;
  operands op_0;
};
op mv_sp_j {                     // J <-- SP
  map mv_sp_j: 38;
  +0[7:0]=38;
  operands op_0;
};
op mv_i_a {                      // A <-- I
  map mv_i_a: 39;
  +0[7:0]=39;
  operands op_0;
};
op mv_i_b {                      // B <-- I
  map mv_i_b: 40;
  +0[7:0]=40;
  operands op_0;
};
op mv_i_fl {                     // FL <-- I
  map mv_i_fl: 41;
  +0[7:0]=41;
  operands op_0;
```

```
};
op mv_i_sp {                    // SP <-- I
  map mv_i_sp: 42;
  +0[7:0]=42;
  operands op_0;
};
op mv_i_mdr {                   // MDR <-- I
  map mv_i_mdr: 43;
  +0[7:0]=43;
  operands op_0;
};
op mv_i_j {                     // J <-- I
  map mv_i_j: 44;
  +0[7:0]=44;
  operands op_0;
};
op mv_j_a {                     // A <-- J
  map mv_j_a: 45;
  +0[7:0]=45;
  operands op_0;
};
op mv_j_b {                     // B <-- J
  map mv_j_b: 46;
  +0[7:0]=46;
  operands op_0;
};
op mv_j_fl {                    // FL <-- J
  map mv_j_fl: 47;
  +0[7:0]=47;
  operands op_0;
};
op mv_j_sp {                    // SP <-- J
  map mv_j_sp: 48;
  +0[7:0]=48;
```

```
  operands op_0;
};
op mv_j_i {                    // I <-- J
  map mv_j_i: 49;
  +0[7:0]=49;
  operands op_0;
};
op mv_j_mdr {                  // MDR <-- J
  map mv_j_mdr: 50;
  +0[7:0]=50;
  operands op_0;
};
op jump {                      // PC <-- arg
  map jump: 51;
  +0[7:0]=51;
  operands op_1;
};
op jump_c {                    // if carry, PC <-- arg
  map jump_c: 52;
  +0[7:0]=52;
  operands op_1;
};
op jump_nc {                   // if not carry, PC <-- arg
  map jump_nc: 53;
  +0[7:0]=53;
  operands op_1;
};
op jump_z {                    // if zero, PC <-- arg
  map jump_z: 54;
  +0[7:0]=54;
  operands op_1;
};
op jump_nz {                   // if not zero, PC <-- arg
  map jump_nz: 55;
```

```
  +0[7:0]=55;
  operands op_1;
};
op jump_n {                    // if negative, PC <-- arg
  map jump_n: 56;
  +0[7:0]=56;
  operands op_1;
};
op jump_nn {                   // if not negative, PC <-- arg
  map jump_nn: 57;
  +0[7:0]=57;
  operands op_1;
};
op jump_o {                    // if overflow, PC <-- arg
  map jump_o: 58;
  +0[7:0]=58;
  operands op_1;
};
op jump_no {                   // if not overflow, PC <-- arg
  map jump_no: 59;
  +0[7:0]=59;
  operands op_1;
};
op call {
  map call : 60;
  +0[7:0]=60;
  operands op_1;
};
op call_i {                    // call I
  map call_i: 61;
  +0[7:0]=61;
  operands op_0;
};
op call_j {                    // call J
```

```
  map call_j: 62;
  +0[7:0]=62;
  operands op_0;
};
op return {
  map return : 63;
  +0[7:0]=63;
  operands op_0;
};
op push_mdr {
  map push_mdr: 64;
  +0[7:0]=64;
  operands op_0;
};
op push_a {
  map push_a: 65;
  +0[7:0]=65;
  operands op_0;
};
op push_b {
  map push_b: 66;
  +0[7:0]=66;
  operands op_0;
};
op push_fl {
  map push_fl: 67;
  +0[7:0]=67;
  operands op_0;
};
op push_i {
  map push_i: 68;
  +0[7:0]=68;
  operands op_0;
};
```

```
op push_j {
  map push_j: 69;
  +0[7:0]=69;
  operands op_0;
};
op pop_mdr {
  map pop_mdr: 70;
  +0[7:0]=70;
  operands op_0;
};
op pop_a {
  map pop_a: 71;
  +0[7:0]=71;
  operands op_0;
};
op pop_b {
  map pop_b: 72;
  +0[7:0]=72;
  operands op_0;
};
op pop_fl {
  map pop_fl: 73;
  +0[7:0]=73;
  operands op_0;
};
op pop_i {
  map pop_i: 74;
  +0[7:0]=74;
  operands op_0;
};
op pop_j {
  map pop_j: 75;
  +0[7:0]=75;
  operands op_0;
```

```
};
op not {
  map not: 76;
  +0[7:0]=76;
  operands op_0;
};
op and {
  map and: 77;
  +0[7:0]=77;
  operands op_0;
};
op or {
  map or: 78;
  +0[7:0]=78;
  operands op_0;
};
op xor {
  map xor: 79;
  +0[7:0]=79;
  operands op_0;
};
op lshl {
  map lshl: 80;
  +0[7:0]=80;
  operands op_0;
};
op lshr {
  map lshr: 81;
  +0[7:0]=81;
  operands op_0;
};
op ashl {
  map ashl: 82;
  +0[7:0]=82;
```

```
  operands op_0;
};
op ashr {
  map ashr: 83;
  +0[7:0]=83;
  operands op_0;
};
op rotl {
  map rotl: 84;
  +0[7:0]=84;
  operands op_0;
};
op rotr {
  map rotr: 85;
  +0[7:0]=85;
  operands op_0;
};
op rotcl {
  map rotcl: 86;
  +0[7:0]=86;
  operands op_0;
};
op rotcr {
  map rotcr: 87;
  +0[7:0]=87;
  operands op_0;
};
op add_c {
  map add_c: 88;
  +0[7:0]=88;
  operands op_0;
};
op sub_b {
  map sub_b: 89;
```

```
  +0[7:0]=89;
  operands op_0;
};
op add {
  map add: 90;
  +0[7:0]=90;
  operands op_0;
};
op sub {
  map sub: 91;
  +0[7:0]=91;
  operands op_0;
};
op in {
  map in : 92;
  +0[7:0]=92;
  operands op_1;
};
op out {
  map out: 93;
  +0[7:0]=93;
  operands op_1;
};
op is_ack {
  map is_ack: 94;
  +0[7:0]=94;
  operands op_2;
};
op stop {
  map stop : 255;
  +0[7:0]=255;
  operands op_0;
};
```

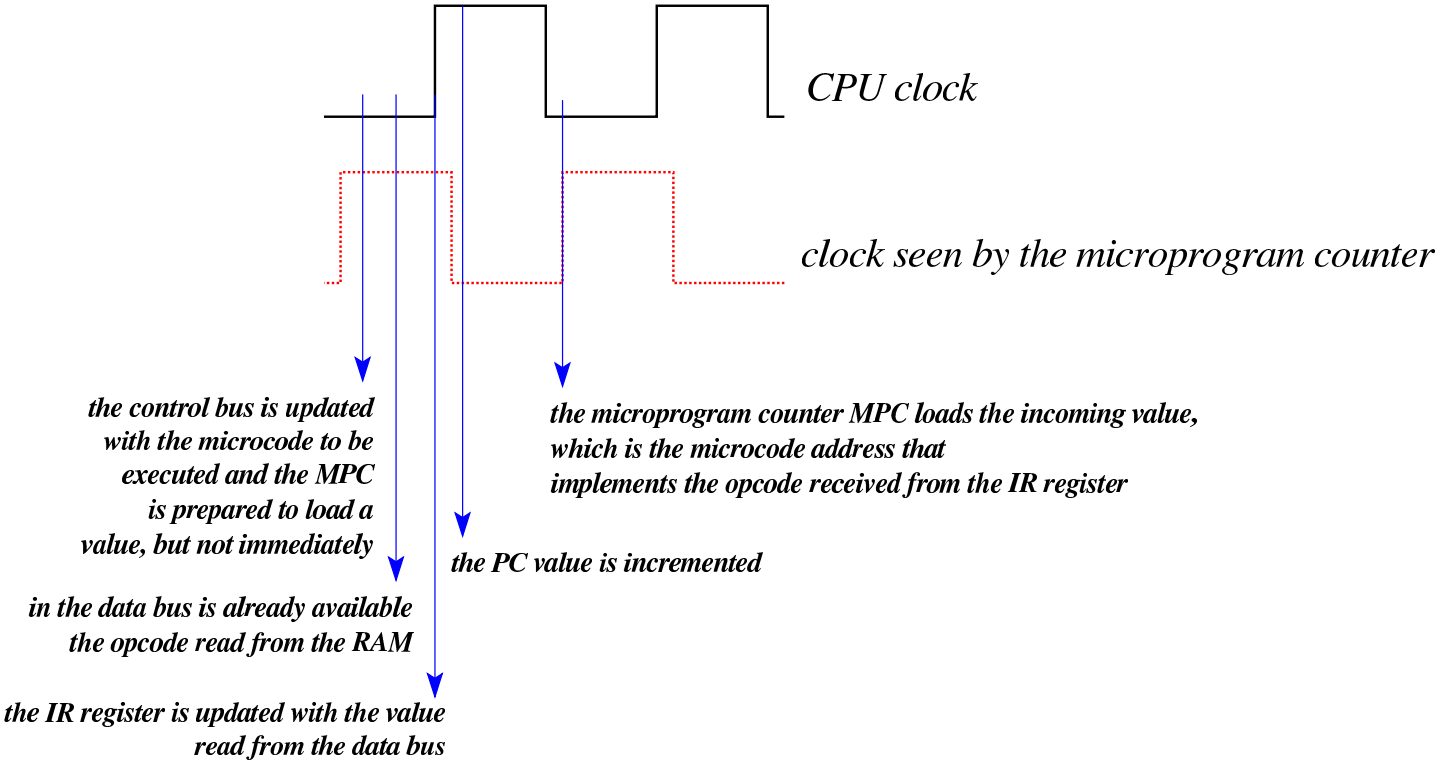
9.8 Microcode

Inside the microcode description section, the first instructions are used to load the first opcode, but in this version, only one clock cycle is needed:

```
begin microcode @ 0
//
    ir=br ram=bw ram=p pc=inc ctrl=load;    // IR <- RAM[pc++],
                                            // jump MAP[ir];
```

The ***IR*** register loads from the data bus what is output from the **RAM** module, which in turn receives the address from the ***PC*** register, which is also post-incremented. In addition to this, it is required to the control unit to update its microprogram-counter (**MPC**) with the value coming from the memory ***ctrl.map***, associated to the address that represents the opcode. All this can be done in a single clock cycle because the structure of the CPU has changed from the previous version. Should be considered the different phases of the clock cycle, which intervene in different ways in the control unit with respect to the components that are then connected to the control bus, as shown in the next picture.

Figure 9.22. Fetch clock cycle.



Therefore, with only one clock cycle, it takes place what is known as fetch. In the

following instructions description, at the end of each procedure the fetch microinstruction is repeated, without the need to restart the microprogram counter **MPC** to the first microinstruction, as it was necessary in previous versions of the simple CPU. In practice, for the macroinstruction **nop** there is only the fetch microinstruction:

```
nop:
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
```

It follows the description of the other macroinstructions:

```
load:
    i=br ram=bw ram=p pc=inc;                     // I <- RAM[pc++];
    mdr=br ram=bw ram=i;                          // MDR <- RAM[i];
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
load_i:
    mdr=br ram=bw ram=i;                          // MDR <- RAM[i];
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
load_j:
    mdr=br ram=bw ram=j;                          // MDR <- RAM[j];
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
store:
    i=br ram=bw ram=p pc=inc;                     // I <- RAM[pc++];
    ram=br ram=i mdr=bw;                          // RAM[i] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
store_i:
    ram=br ram=i mdr=bw;                          // RAM[i] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
store_j:
    ram=br ram=j mdr=bw;                          // RAM[j] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
cp_ij:
    mdr=br ram=bw ram=i i=inc;                    // MDR <- RAM[i++];
    ram=br ram=j mdr=bw j=inc;                    // RAM[j++] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
cp_ji:
    mdr=br ram=bw ram=j j=inc;                    // MDR <- RAM[j++];
    ram=br ram=i mdr=bw i=inc;                    // RAM[i++] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
mv_mdr_a:
    a=br mdr=bw;                                  // A <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;          // fetch
mv_mdr_b:
    b=br mdr=bw;                                  // B <- MDR;
```

```
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_fl:
    fl=br mdr=bw;                               // FL <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_sp:
    sp=br mdr=bw;                               // SP <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_i:
    i=br mdr=bw;                                // I <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_j:
    j=br mdr=bw;                                // J <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_mdr:
    mdr=br a=bw;                                // MDR <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_b:
    b=br a=bw;                                  // B <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_fl:
    fl=br a=bw;                                 // FL <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_sp:
    sp=br a=bw;                                 // SP <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_i:
    i=br  a=bw;                                 // I <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_j:
    j=br  a=bw;                                 // J <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_a:
    a=br b=bw;                                  // A <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_mdr:
    mdr=br b=bw;                                // MDR <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_fl:
    fl=br b=bw;                                 // FL <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_sp:
    sp=br b=bw;                                 // SP <- B;
```

```
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_i:
    i=br b=bw;                                  // I <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_j:
    j=br b=bw;                                  // J <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_a:
    a=br fl=bw;                                 // A <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_b:
    b=br fl=bw;                                 // B <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_mdr:
    mdr=br fl=bw;                               // MDR <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_sp:
    sp=br fl=bw;                                // SP <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_i:
    i=br fl=bw;                                 // I <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_j:
    j=br fl=bw;                                 // J <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_a:
    a=br sp=bw;                                 // A <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_b:
    b=br sp=bw;                                 // B <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_fl:
    fl=br sp=bw;                                // FL <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_mdr:
    mdr=br sp=bw;                               // MDR <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_i:
    i=br sp=bw;                                 // I <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_j:
    j=br sp=bw;                                 // J <- SP;
```

```
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_a:
    a=br i=bw;                                  // A <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_b:
    b=br i=bw;                                  // B <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_fl:
    fl=br i=bw;                                 // FL <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_sp:
    sp=br i=bw;                                 // SP <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_mdr:
    mdr=br i=bw;                                // MDR <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_j:
    j=br i=bw;                                  // J <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_a:
    a=br j=bw;                                  // A <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_b:
    b=br j=bw;                                  // B <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_fl:
    fl=br j=bw;                                 // FL <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_sp:
    sp=br j=bw;                                 // SP <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_i:
    i=br j=bw;                                  // I <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_mdr:
    mdr=br j=bw;                                // MDR <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump:
    i=br pc=bw;                                 // I <- PC
    pc=br ram=bw ram=i;                         // PC <-- RAM[i]
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_c:
```

```
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_carry;                         // PC = (carry?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_nc:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_not_carry;                     // PC = (not_carry?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_z:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_zero;                          // PC = (zero?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_nz:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_not_zero;                      // PC = (not_zero?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_n:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_negative;                      // PC = (negative?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_nn:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_not_negative;                  // PC = (not_negative?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_o:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_overflow;                      // PC = (overflow?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_no:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_not_overflow;                  // PC = (not_overflow?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
call:
    i=br ram=bw ram=p pc=inc sp=dec;            // I <- RAM[pc++], SP--;
    ram=br ram=s pc=bw;                         // RAM[sp] <- PC;
    pc=br i=bw;                                 // PC <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
call_i:
    sp=dec;                                     // SP--;
    ram=br ram=s pc=bw;                         // RAM[sp] <- PC;
    pc=br i=bw;                                 // PC <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
call_j:
```

```
    sp=dec;                                          // SP--;
    ram=br ram=s pc=bw;                              // RAM[sp] <- PC;
    pc=br j=bw;                                      // PC <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
return:
    pc=br ram=bw ram=s sp=inc;                       // PC <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
push_mdr:
    sp=dec;                                          // SP--;
    ram=br ram=s mdr=bw;                             // RAM[sp] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
push_a:
    sp=dec;                                          // SP--;
    ram=br ram=s a=bw;                               // RAM[sp] <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
push_b:
    sp=dec;                                          // SP--;
    ram=br ram=s b=bw;                               // RAM[sp] <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
push_fl:
    sp=dec;                                          // SP--;
    ram=br ram=s fl=bw;                              // RAM[sp] <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
push_i:
    sp=dec;                                          // SP--;
    ram=br ram=s i=bw;                               // RAM[sp] <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
push_j:
    sp=dec;                                          // SP--;
    ram=br ram=s j=bw;                               // RAM[sp] <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
pop_mdr:
    mdr=br ram=bw ram=s sp=inc;                      // MDR <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
pop_a:
    a=br ram=bw ram=s sp=inc;                        // A <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
pop_b:
    b=br ram=bw ram=s sp=inc;                        // B <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;             // fetch
pop_fl:
    fl=br ram=bw ram=s sp=inc;                       // FL <- RAM[sp++];
```

```
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
pop_i:
    i=br ram=bw ram=s sp=inc;                   // I <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
pop_j:
    j=br ram=bw ram=s sp=inc;                   // J <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
not:
    a=br alu=not fl=xr;                         // A <- NOT A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
and:
    a=br alu=and fl=xr;                         // A <- A AND B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
or:
    a=br alu=or fl=xr;                          // A <- A OR B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
xor:
    a=br alu=xor fl=xr;                         // A <- A XOR B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
lshl:
    a=br alu=lshl fl=xr;                        // A <- A << 1
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
lshr:
    a=br alu=lshr fl=xr;                        // A <- A >> 1
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
ashl:
    a=br alu=ashl fl=xr;                        // A <- A*2
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
ashr:
    a=br alu=ashr fl=xr;                        // A <- A/2
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
rotl:
    a=br alu=rotl fl=xr;                        // A <- rotl(A)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
rotr:
    a=br alu=rotr fl=xr;                        // A <- rotr(A)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
rotcl:
    a=br alu=rotcl fl=xr;                       // A <- rotcl(A)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
rotcr:
    a=br alu=rotcr fl=xr;                       // A <- rotcr(A)
```

```
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
add_c:
    a=br alu=add_c fl=xr;                       // A <- A+B+carry
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
sub_b:
    a=br alu=sub_b fl=xr;                       // A <- A-B-borrow
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
add:
    a=br alu=add fl=xr;                         // A <- A+B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
sub:
    a=br alu=sub fl=xr;                         // A <- A-B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
in:
    ioa=br ram=bw ram=p pc=inc;                 // IOA <- RAM[pc++];
    ioc=req;                                    // I/O request;
    ctrl=nop;                                   // does not do anything
    a=br ioc=bw;                                // A <- I/O
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
out:
    ioa=br ram=bw ram=p pc=inc;                 // IOA <- RAM[pc++];
    ioc=br a=bw;                                // I/O <- A
    ioc=req;                                    // I/O request;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
is_ack:
    ioa=br ram=bw ram=p pc=inc;                 // IOA <- RAM[pc++];
    mdr=br ram=bw ram=p pc=inc;                 // MDR <- RAM[pc++];
    a=br ioc=bw ioc=isack;                      // A <- I/O is ack;
    a=br alu=not fl=xr;                         // A <- NOT A;
    a=br alu=not fl=xr;                         // A <- NOT A;
    pc=br sel=if_not_zero;                      // PC = (not_zero?MDR:PC);
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
stop:
    ctrl=stop;                                  // stop clock
    // if resumed:
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
end
```

The set of macroinstructions has changed slightly and is also extended, taking into account the changes to the CPU, as described in the following table.

Table 9.25. Macroinstruction list for the current simple CPU.

Syntax	Clock cycles	Description
`nop`	1	Do nothing: just fetch the next instruction.
`load` ***address***	3	The instruction loads the argument into the ***I*** register and then loads into the ***MDR*** register the memory content at the specified address.
`load_i` `load_j`	2	The instruction loads into the ***MDR*** register the memory content at the address specified by the ***I*** or ***J*** registers.
`store` ***address***	3	The instruction loads the argument into the ***I*** register and then it writes the ***MDR*** content into the memory, at the address specified.
`store_i` `store_j`	2	The instruction writes the ***MDR*** content into memory, at the address specified by the ***I*** or ***J*** registers.
`cp_ij` `cp_ji`	3	The first instruction copies the memory content from the address specified by the ***I*** register to the address specified by the ***J*** register. The second instruction does the opposite.

Syntax	Clock cycles	Description

`mv_mdr_a`
`mv_mdr_b`
`mv_mdr_fl`
`mv_mdr_sp`
`mv_mdr_i`
`mv_mdr_j`
	2	Copy the content of the ***MDR*** register to the register ***A***, ***B***, ***FL***, ***SP***, ***I*** or ***J***.

`mv_a_mdr`
`mv_a_b`
`mv_a_fl`
`mv_a_sp`
`mv_a_i`
`mv_a_j`
	2	Copy the content of the ***A*** register to the register ***MDR***, ***B***, ***FL***, ***SP***, ***I*** or ***J***.

Syntax	Clock cycles	Description

`mv_b_a`
`mv_b_mdr`
`mv_b_fl`
`mv_b_sp`
`mv_b_i`
`mv_b_j`
	2	Copy the content of the ***B*** register to the register ***A***, ***MDR***, ***FL***, ***SP***, ***I*** or ***J***.

`mv_fl_a`
`mv_fl_b`
`mv_fl_mdr`
`mv_fl_sp`
`mv_fl_i`
`mv_fl_j`
	2	Copy the content of the ***FL*** register to the register ***A***, ***B***, ***MDR***, ***SP***, ***I*** or ***J***.

Syntax	Clock cycles	Description

`mv_sp_a`
`mv_sp_b`
`mv_sp_fl`
`mv_sp_mdr`
`mv_sp_i`
`mv_sp_j`
	2	Copy the content of the ***SP*** register to the register ***A***, ***B***, ***FL***, ***MDR***, ***I*** or ***J***.

`mv_i_a`
`mv_i_b`
`mv_i_fl`
`mv_i_sp`
`mv_i_mdr`
`mv_i_j`
	2	Copy the content of the ***I*** register to the register ***A***, ***B***, ***FL***, ***SP***, ***MDR*** or ***J***.

Syntax	Clock cycles	Description

`mv_j_a`
`mv_j_b`
`mv_j_fl`
`mv_j_sp`
`mv_j_i`
`mv_j_mdr`
	2	Copy the content of the ***J*** register to the register ***A***, ***B***, ***FL***, ***SP***, ***I*** or ***MDR***.

`jump` ***address***	3	Jump to the specified address: it puts the argument into the ***I*** register and then jumps to the instruction at that address.
`jump_c` ***address*** `jump_nc` ***address*** `jump_z` ***address*** `jump_nz` ***address*** `jump_n` ***address*** `jump_nn` ***address*** `jump_o` ***address*** `jump_no` ***address***	3	These instructions put the argument inside the ***MDR*** register, then, if the condition implied by the name is true, they jumps to the instruction at the address specified. The conditions are in this order: carry, not carry, zero, not zero, negative, not negative, overflow, not overflow.

Syntax	Clock cycles	Description
`call` ***address***	4	Call a procedure: it puts the argument into the ***I*** register and it decrement the ***SP*** register, then, it saves the ***PC*** register to the memory address contained inside the ***SP*** register (it is a push). Then it jumps to the address specified.
`call_i` `call_j`	4	Call a procedure: it decrement the ***SP*** register, then, it saves the ***PC*** register to the memory address contained inside the ***SP*** register (it is a push). Then it jumps to the address specified by the register ***I*** or ***J***.
`return`	2	Return from procedure call: it reads the value contained in memory at the ***SP*** position and it writes that value to the ***PC*** register, then increment the ***SP*** register.

`push_mdr`
`push_a`
`push_b`
`push_fl`
`push_sp`
`push_i`
`push_j`
	3	Save a register on to the stack: it decrements the ***SP*** register, then, at the new address specified by the ***SP*** register, it writes in memory the value of the register named as an extention to the instruction name. That is: ***MDR***, ***A***, ***B***, ***FL***, ***SP***, ***I*** or ***J***.

Syntax	Clock cycles	Description

`pop_mdr`
`pop_a`
`pop_b`
`pop_fl`
`pop_sp`
`pop_i`
`pop_j`
	3	Restore a register from the stack: it copies inside the specified register the memory content at the position specified by the ***SP*** register, then it increments the same ***SP*** register.

`not`	2	***A*** = ~***A***

`and`
`or`
`xor`
	2	***A*** = ***A*** & ***B***
		A = ***A*** | ***B***
		A = ***A*** ^ ***B***

`lshl` `lshr`	2	***A*** = left or right logic shift of ***A***
`ashl` `ashr`	2	***A*** = left or right arithmetic shift of ***A***

Syntax	Clock cycles	Description
`rotl` `rotl`	2	***A*** = left or right rotation of ***A***
`rotcl` `rotcl`	2	***A*** = left or right rotation with carry of ***A***

`add_c`
 	
`sub_b`
	2	***A = A+B+carry***
		A = A−B−carry

`add`
 	
`sub`
	2	***A = A+B***
		A = A−B

`in` ***io_address***	5	Read from I/O device: it read from the I/O address specified and the result is placed inside register ***A***.
`out` ***io_address***	4	Write to an I/O device: it writes the content of register ***A*** to the I/O device with the address specified.
`is_ack` ***io_address address***	7	Jump if previous I/O operation is acknowledged: it load the specified I/O address into the ***MDR*** register and, if the device returns a positive acknowledge, it jumps to the last argument address. The operation destroys the ***A*** register content and updates the flags.
`stop`	1	Halt: it stops the clock.

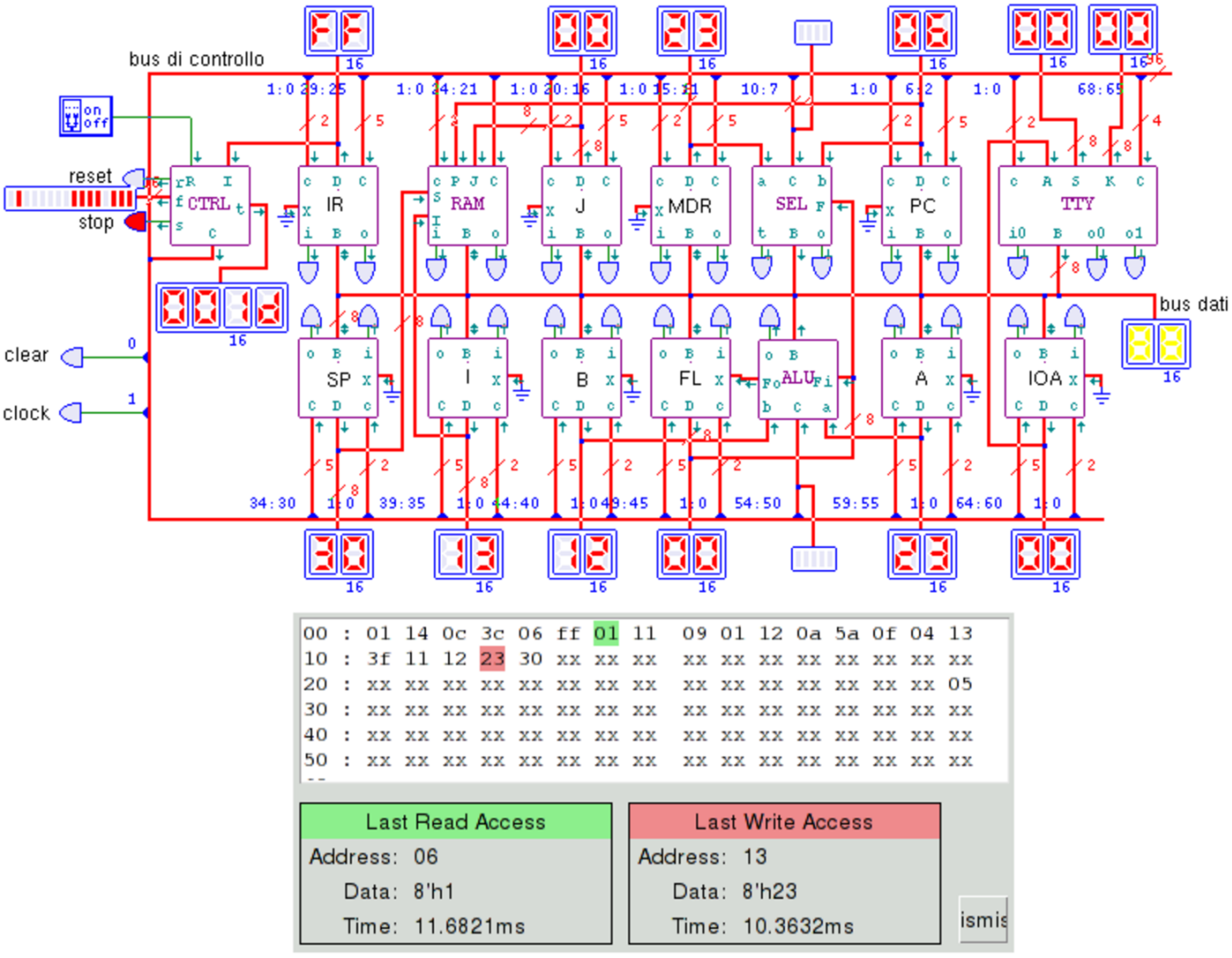
The following sections provide some examples to verify the operation of macroin-

structions with the current version of the simple CPU. The examples are combined with videos where also the RAM access is highlighted.

9.9 Macrocode: routine call

```
begin macrocode @ 0
start:
        load #data_3            // Place the stack.
        mv_mdr_sp
        call #sum               // Call the function sum().
        stop
sum:
        load #data_0
        mv_mdr_a
        load #data_1
        mv_mdr_b
        add
        mv_a_mdr
        store #data_2
        return
data_0:
        .byte 0x11
data_1:
        .byte 0x12
data_2:
        .byte 0x00
data_3:
        .byte 0x30
end
```

Figure 9.27. Data bus after the execution of the code listed above. Video: ogv *http://www.youtube.com/watch?v=eATz3XLYWbc*

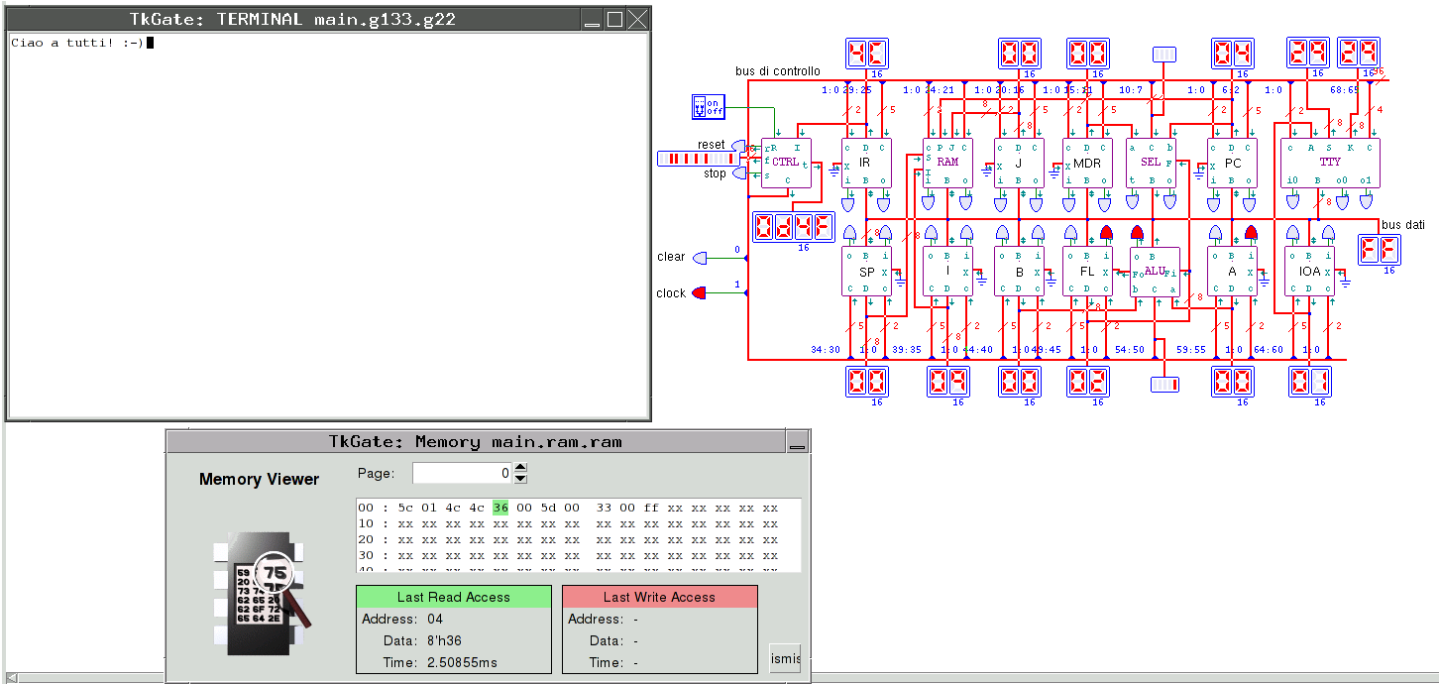


9.10 Macrocode: keyboard input and screen output

```
begin macrocode @ 0
start:
        in 1            // Read from the keyboard.
        not             // Complement two times just to update
        not             //   the status flags.
        jump_z #start   // If the value is zero, then repeat
                        //   the keyboard read.
        out 0           // Otherwise, display the same
                        //   value on screen.
```

```
        jump #start     // Restart.
stop:                   // It will never reach this point.
        stop
end
```

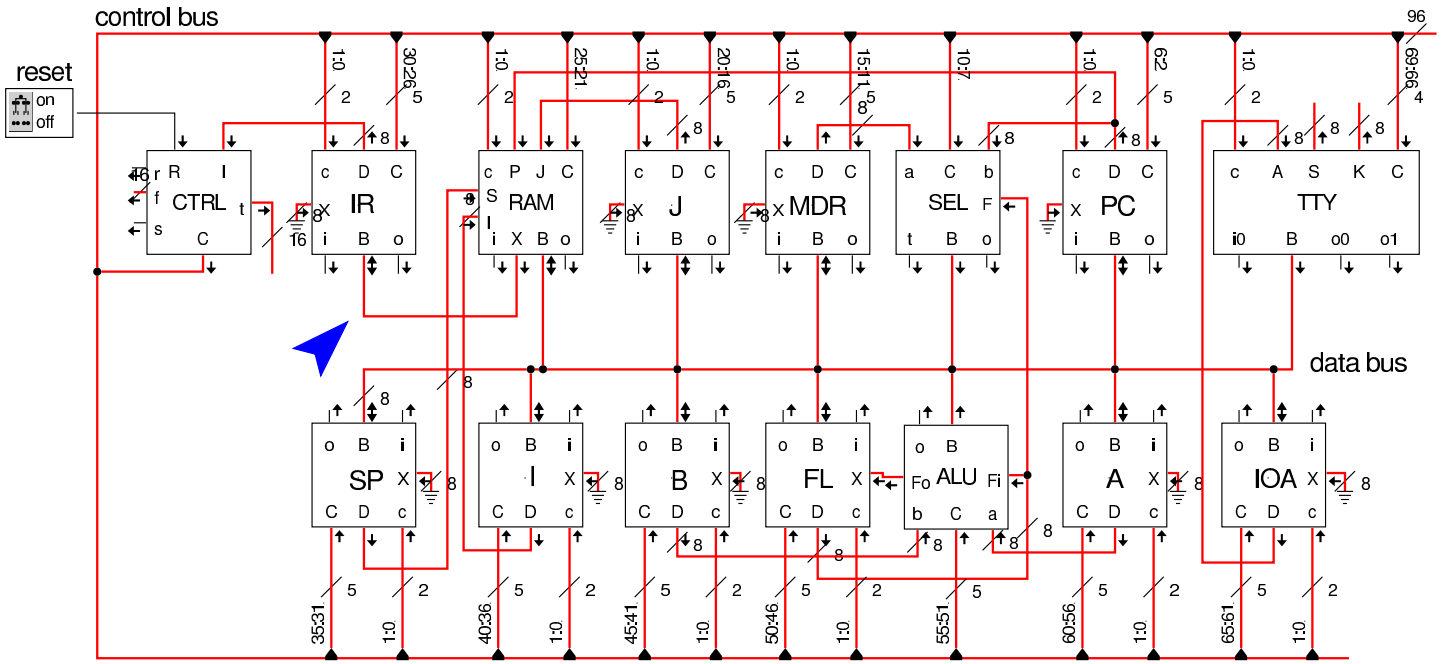
Figure 9.29. Keyboard input and screen output. Video: ogv *http://www.youtube.com/watch?v=m22oK22ULTwWo*



9.11 Version "B2"

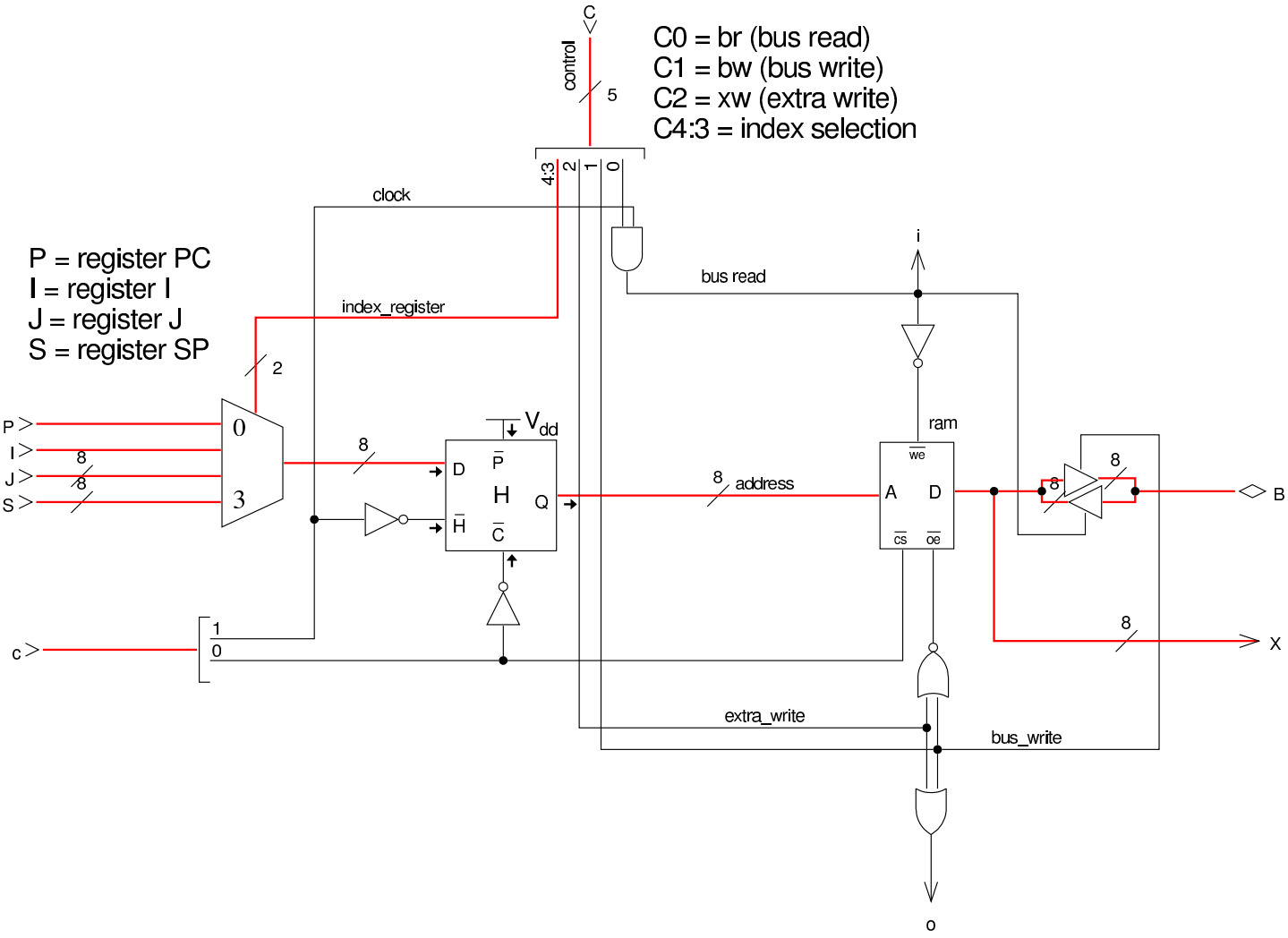
Here is described a further modification to the simple CPU, in order to slightly improve its efficiency through a direct connection between the **RAM** module and the ***IR*** register, so that an opcode can be transferred from the memory to the instruction register without interfering with the data bus.

Figure 9.30. Simple CPU, version "B2".



To make the connection shown in the figure, the **RAM** module is modified, in order to drive an auxiliary output (*X*); consequently, it enlarges its connection to the control bus, forcing to readapt the connections of the other modules.

Figure 9.31. Module **RAM** modified with the addition of the auxiliary output.



The fields declaration of the control word requires a change of meaning of the control lines on the **RAM** module, together with the shift on the connection of the other modules:

```
field ctrl[1:0]   = {nop=0, stop=1, load=2};
field pc[6:2]     = {br=1, bw=2, xr=4, inc=8, dec=16};
field sel[10:7]   = {if_carry=1, if_not_carry=3,
                     if_zero=5, if_not_zero=7,
                     if_negative=9, if_not_negative=11,
                     if_overflow=13, if_not_overflow=15};
field mdr[15:11]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field j[20:16]    = {br=1, bw=2, xr=4, inc=8, dec=16};
field ram[25:21]  = {br=1, bw=2, xw=4, p=0, i=8, j=16, s=24};
field ir[30:26]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field sp[35:31]   = {br=1, bw=2, xr=4, inc=8, dec=16};
```

```
field i[40:36]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field b[45:41]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field fl[50:46]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field alu[55:51] = {not=1, and=3, or=5, xor=7, lshl=9, lshr=11,
                    ashl=13, ashr=15, rotl=17, rotr=19, rotcl=21,
                    rotcr=23, add_c=25, sub_b=27, add=29, sub=31};
field a[60:56]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field ioa[65:61] = {br=1, bw=2, xr=4, inc=8, dec=16};
field ioc[69:66] = {br=1, bw=2, req=4, isack=8};
```

The difference inside the microcode declaration, is due to the fact that the RAM must have enabled the line ***xr*** to be able to communicate with the ***IR*** register, therefore, all the microinstruction for the fetch should be changed in the following way:

```
    ir=br ram=xw ram=p pc=inc ctrl=load;
```

Finally, where possible, the fetch microinstruction that appears at the end of each macroinstruction procedure, is merged with the penultimate microinstruction. It is possible precisely with the instructions for copying a value from one register to another and those that do something with the ALU:

```
mv_mdr_a:
    a=br mdr=bw ir=br ram=xw ram=p pc=inc ctrl=load;
mv_mdr_b:
    b=br mdr=bw ir=br ram=xw ram=p pc=inc ctrl=load;
…
mv_j_i:
    i=br j=bw ir=br ram=xw ram=p pc=inc ctrl=load;
mv_j_mdr:
    mdr=br j=bw ir=br ram=xw ram=p pc=inc ctrl=load;
```

```
not:
    a=br alu=not fl=xr ir=br ram=xw ram=p pc=inc ctrl=load;
and:
    a=br alu=and fl=xr ir=br ram=xw ram=p pc=inc ctrl=load;
```

```
…
add:
    a=br alu=add fl=xr ir=br ram=xw ram=p pc=inc ctrl=load;
sub:
    a=br alu=sub fl=xr ir=br ram=xw ram=p pc=inc ctrl=load;
```

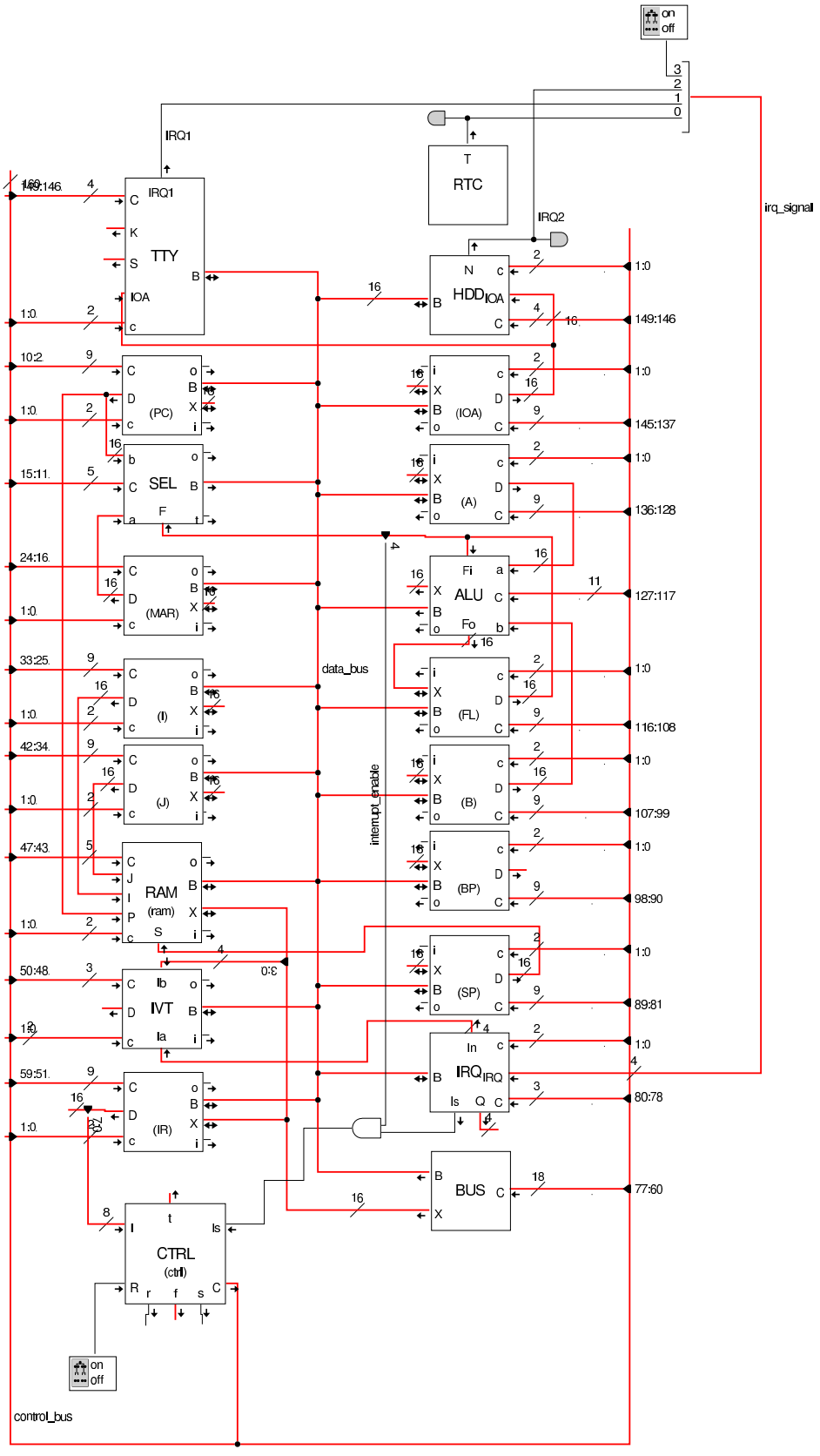
10.1 General purpose 16-bit registers 505
10.2 Module "BUS" 508
10.3 Module "ALU" 509
10.4 Module "SEL" 518
10.5 Module "RAM" 519
10.6 Module "IRQ" 521
10.7 Module "IVT" 529
10.8 Module "CTRL" 531
10.9 Opcodes 534
10.10 Microcode 558
10.11 Interrupts management 578
10.12 Module "RTC": real time clock 580
10.13 Module "TTY" 581
10.14 Module "HDD" 586
10.15 Macrocode: terminal usage example, through interrupts 592

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

Here is a further extension of the project with 16-bit registers, while continuing to manage a memory organized in 8-bit words. Since the TKGate microcode-macrocode compiler stores the 16-bit values by reversing the order of the bytes (or at least it does in the version compiled for x86 architecture), this version of the CPU (which is now a computer with devices) is organized in little-endian mode.

Attachments	Description
attachments/xcpu/xcpu-c.v	TKGate Verilog netlist source file.
attachments/xcpu/xcpu-c.gm	TKGate microcode and macrocode source file.
attachments/xcpu/xcpu-c-terminal.vpd.tcl	TCL/Tk script related to the terminal module **TTY**.

Figure 10.2. Simple CPU version "C".



Among the other new things, in the figure should be observed the presence of the

BP register (*base pointer*) whose purpose is to facilitate the use of the stack of data when performing function calls. The register should be used in a manner similar to the 8086-8088 calling conventions; otherwise, the same register is known with the name "FP" (*frame pointer*).

10.1 General purpose 16-bit registers

The registers of this version of the project have a 16-bit size, but are divided into two bytes, the contents of which can be accessed separately. Furthermore, it is possible to increase and reduce the value of these registers, of one or two units.

Figure 10.3. 16-bit registers: external appearance.

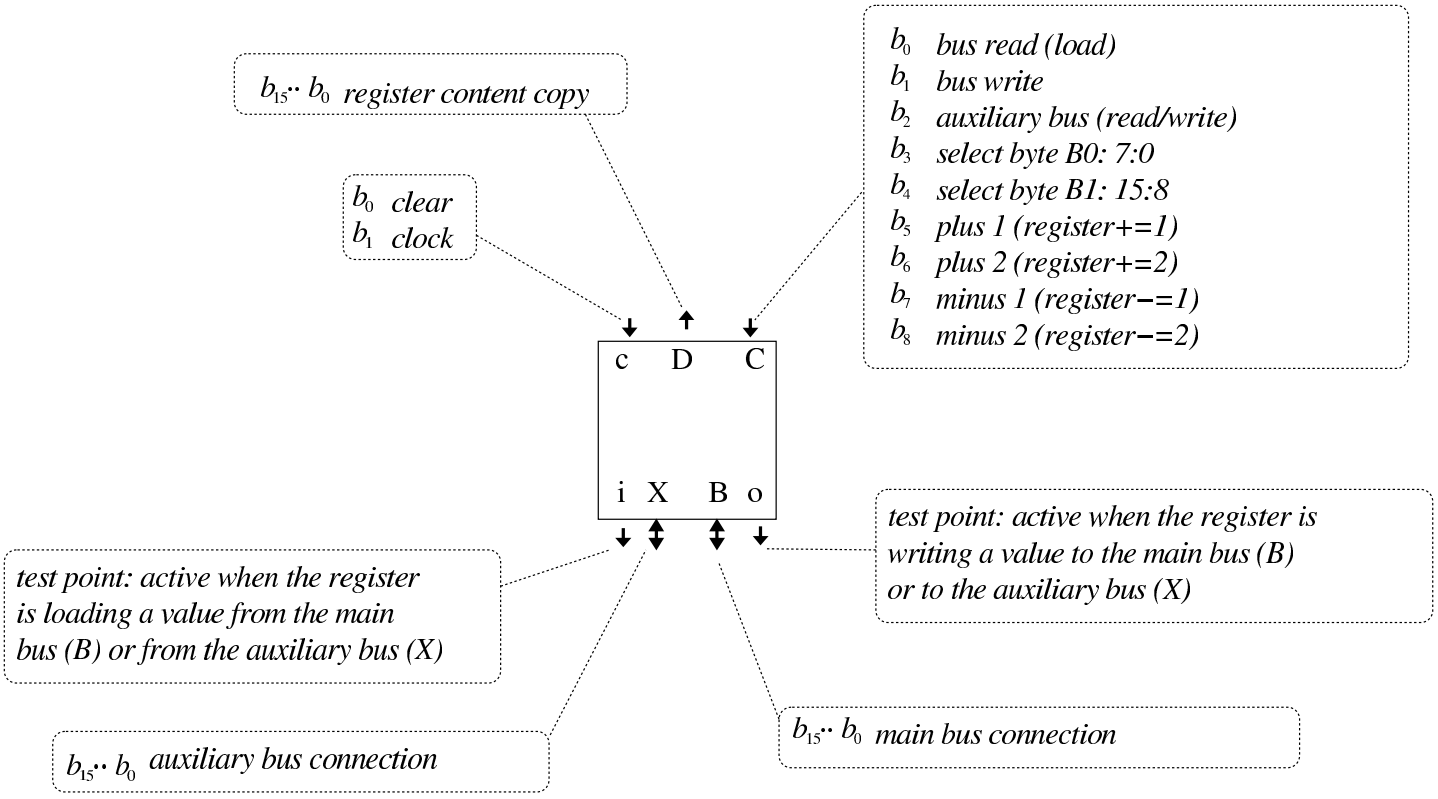


Figure 10.4. 16-bit registers: internal structure.

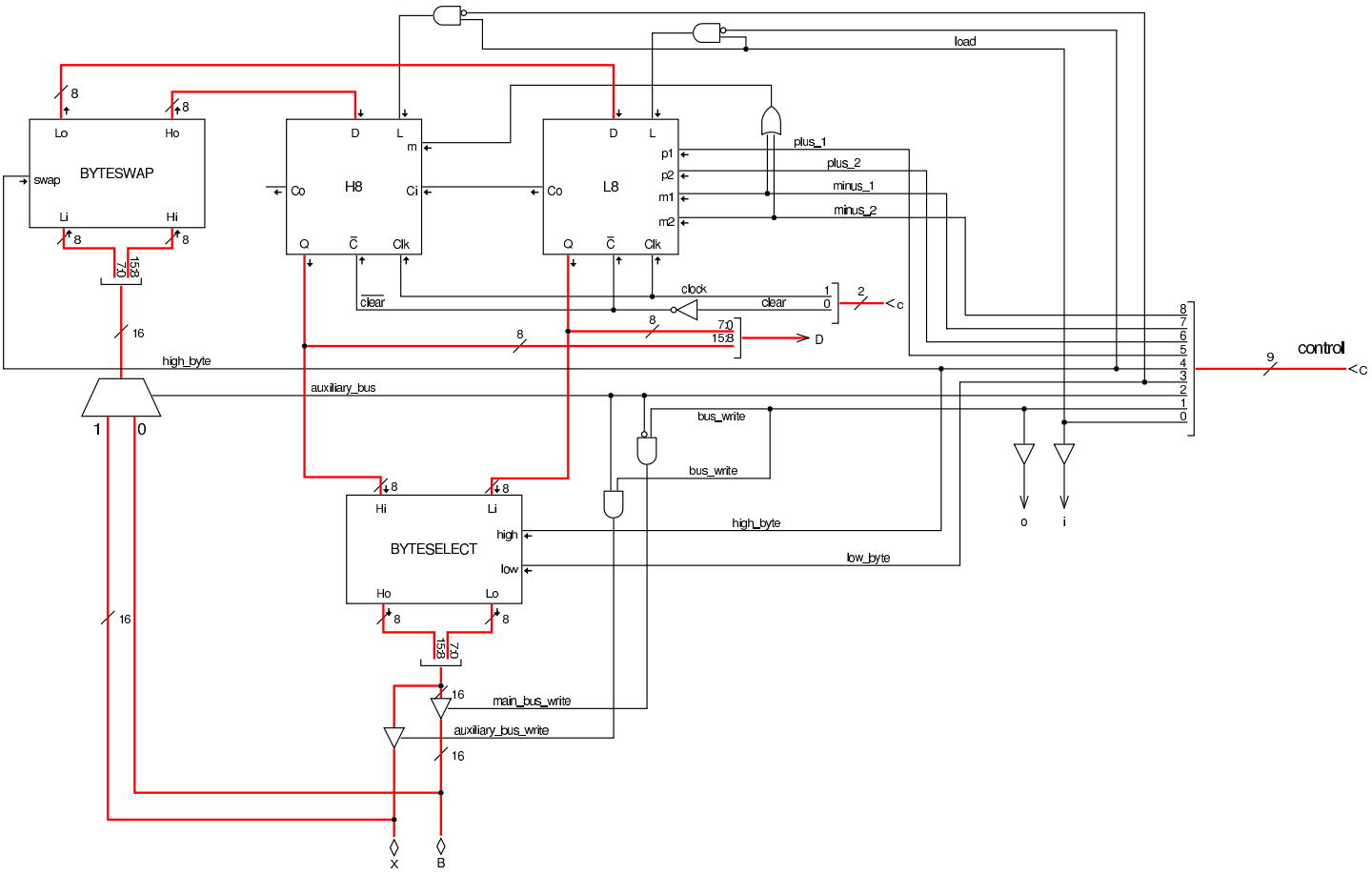
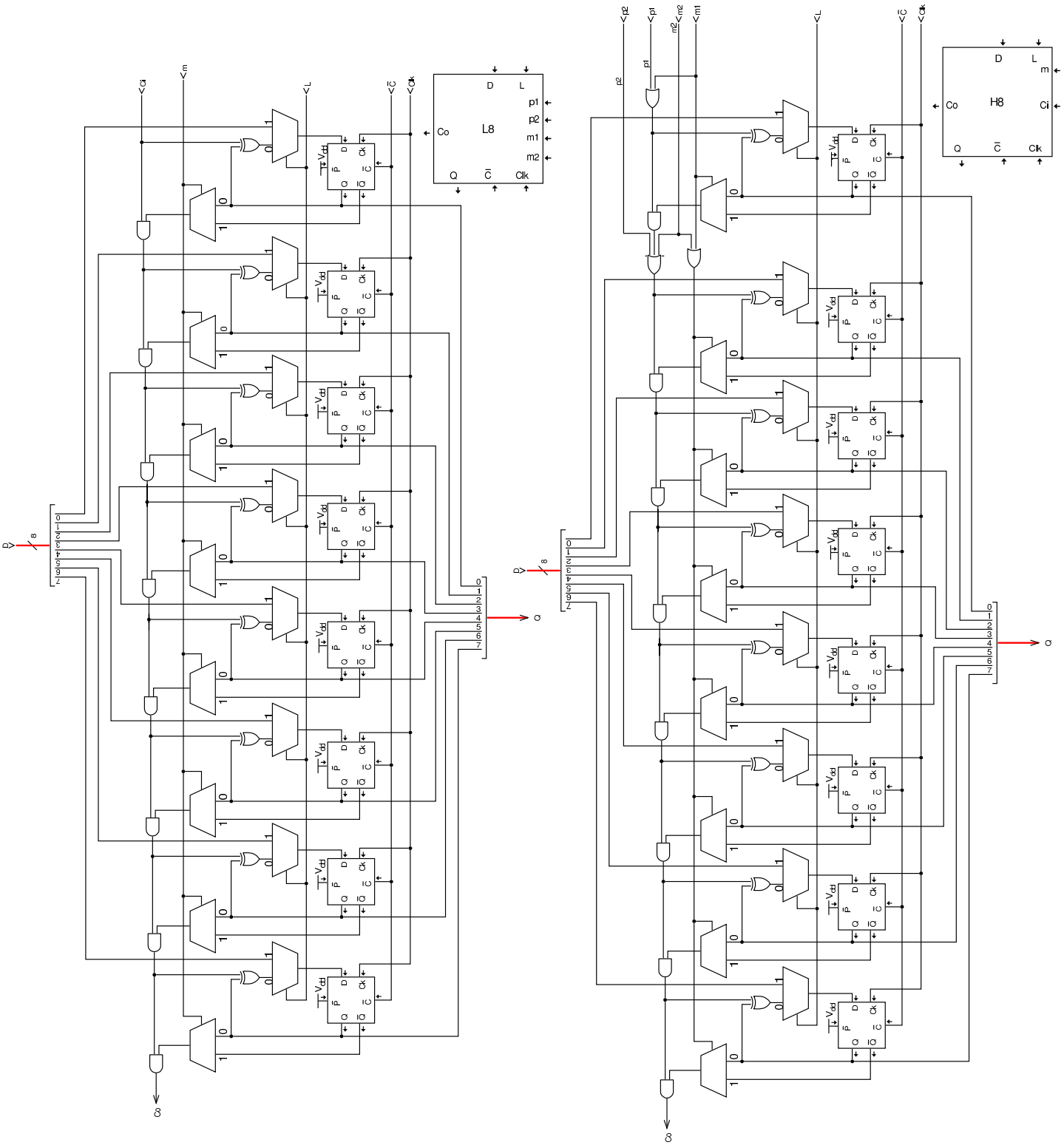


Figure 10.5. Modules **H8** and **L8**.

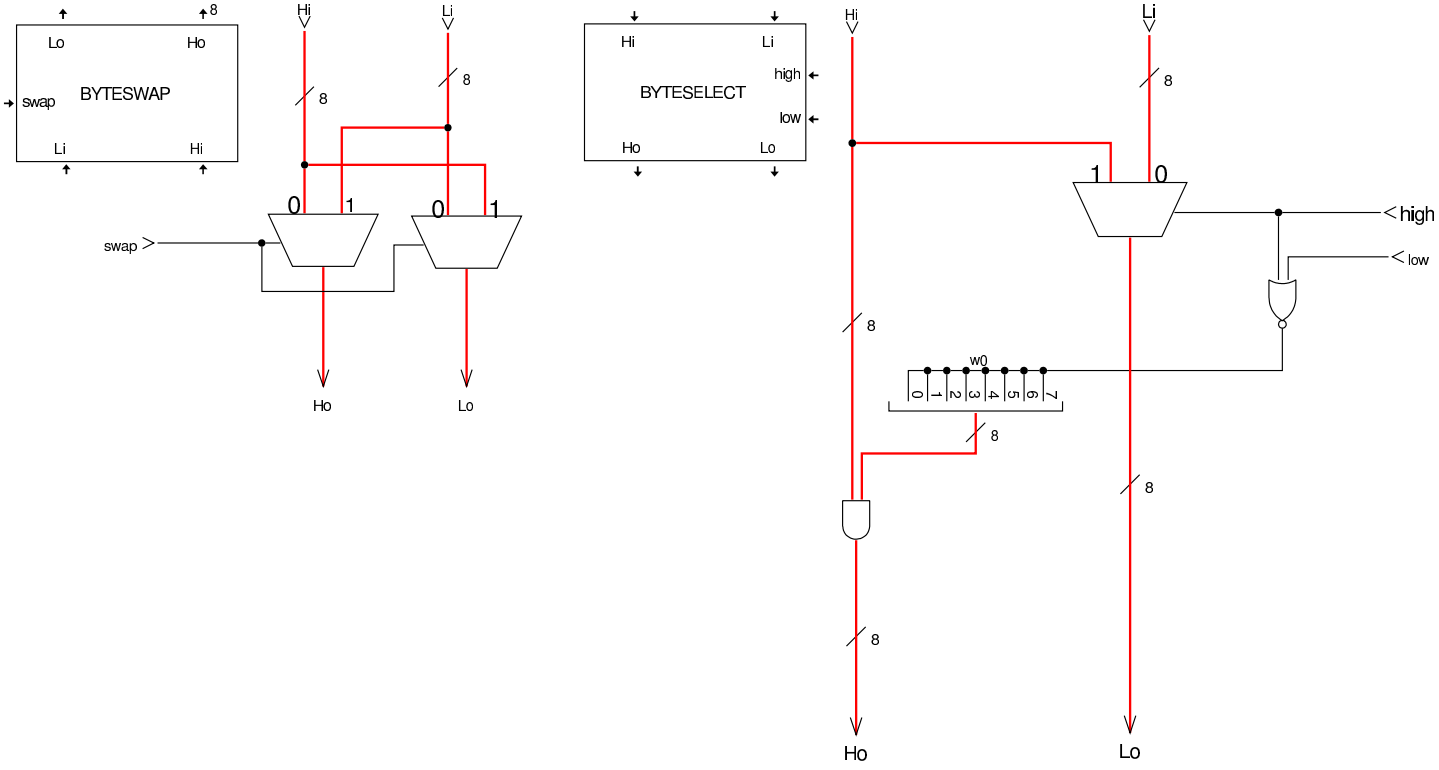


The module **BYTESELECT** contained inside the 16-bit registers, is used to reduce the reading of the contents of the register to only 8 bits, choosing from the least significant byte or the more significant one. For example, if the register value is $ABCD_{16}$ and the

least significant byte is selected, the value $00CD_{16}$ is obtained; on the other hand, if the more significant byte is selected, the value $00AB_{16}$ is obtained.

When a value is to be written inside the register, the separate modules **`H8`** and **`L8`** can load independently. If a single byte is to be written, to the most significant position inside the register (without modifying the least significant one), the input from the bus is swapped by the module **`BYTESWAP`** and the **`H8`** loads alone.

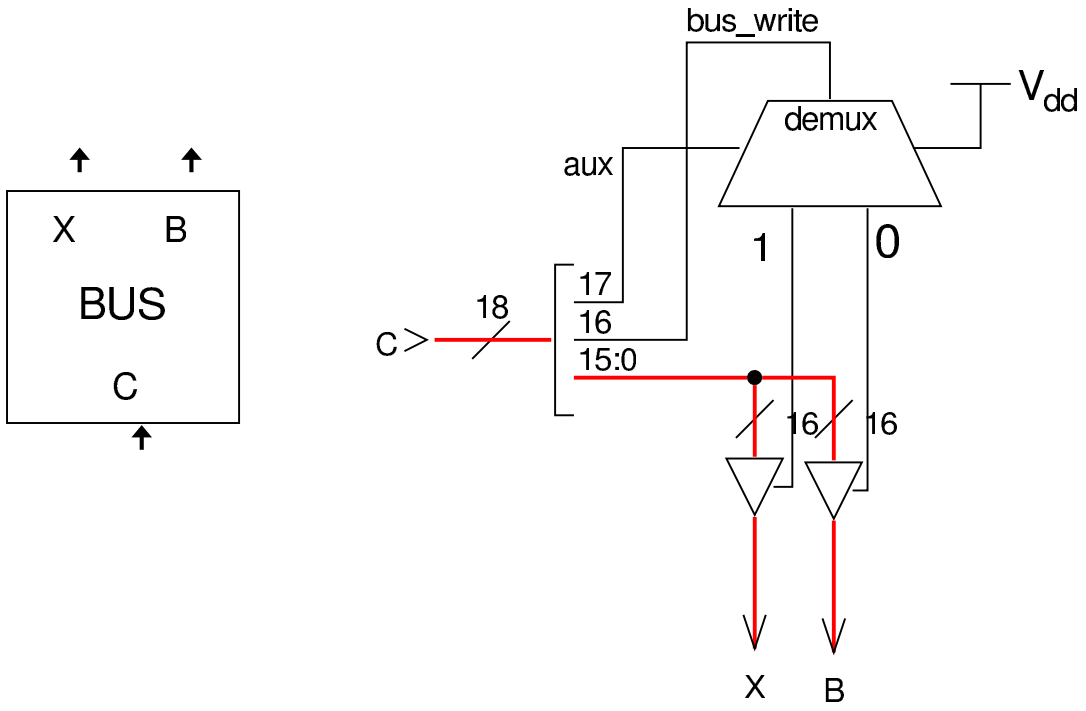
Figure 10.6. Modules **`BYTESWAP`** and **`BYTESELECT`**.



10.2 Module "BUS"

For this version of the project there is a new module that allows to the control unit to write a value to the data bus. The module can select the main bus (***B***) or the auxiliary bus (***X***).

Figure 10.7. Module **BUS**.



10.3 Module "ALU"

The ALU unit is redesigned from the previous version of the project. Currently is it able to handle 16-bit values and some more functions. It should be noticed that there are separated flags for 8-bit and 16-bit operations.

Figure 10.8. ALU overall structure.

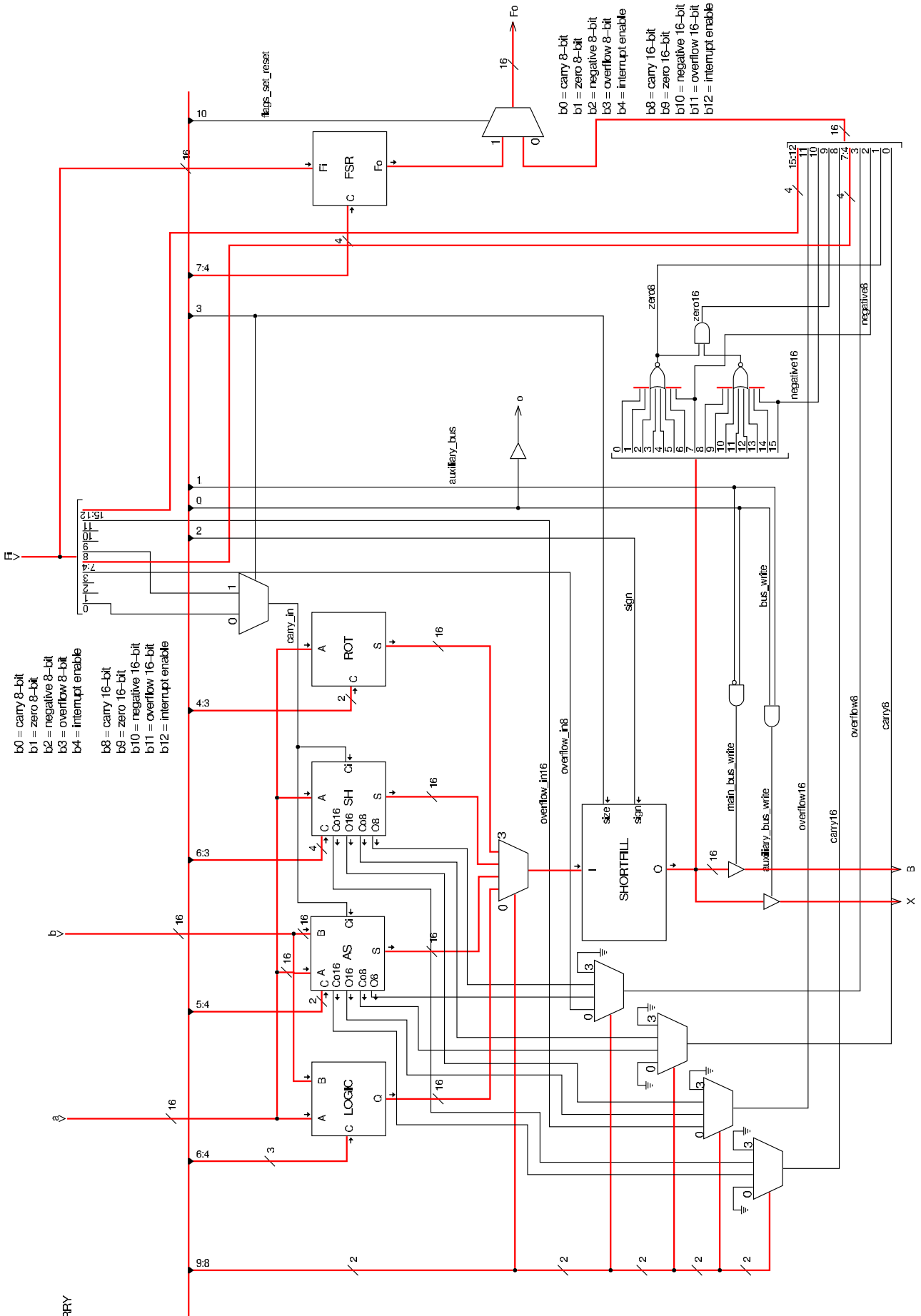
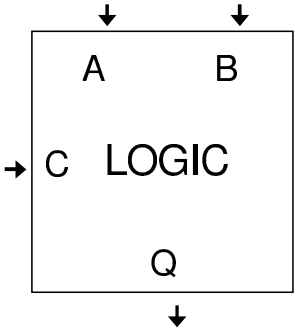


Figure 10.9. Login unit inside the ALU.



C=0: q = a
C=1: q = a AND B
C=2: q = a OR b
C=3: q = a XOR b
C=4: q = NOT (a XOR b)
C=5: q = NOT (a OR b)
C=6: q = NOT (a AND b)
C=7: q = NOT a

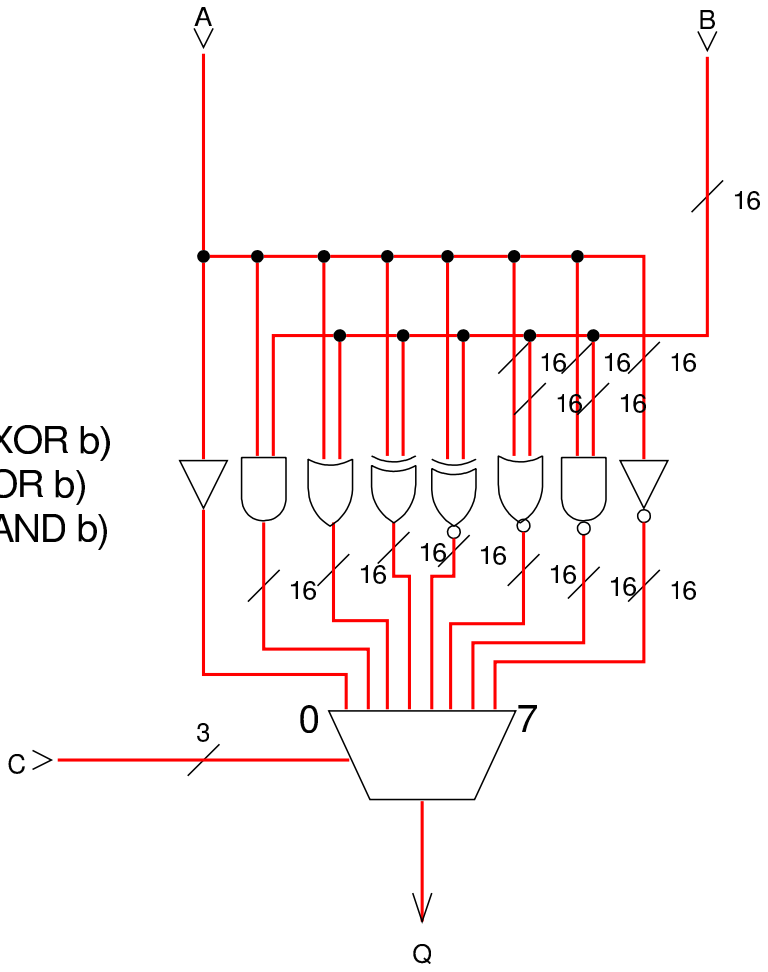


Figure 10.10. Module **AS**: addition and subtraction.

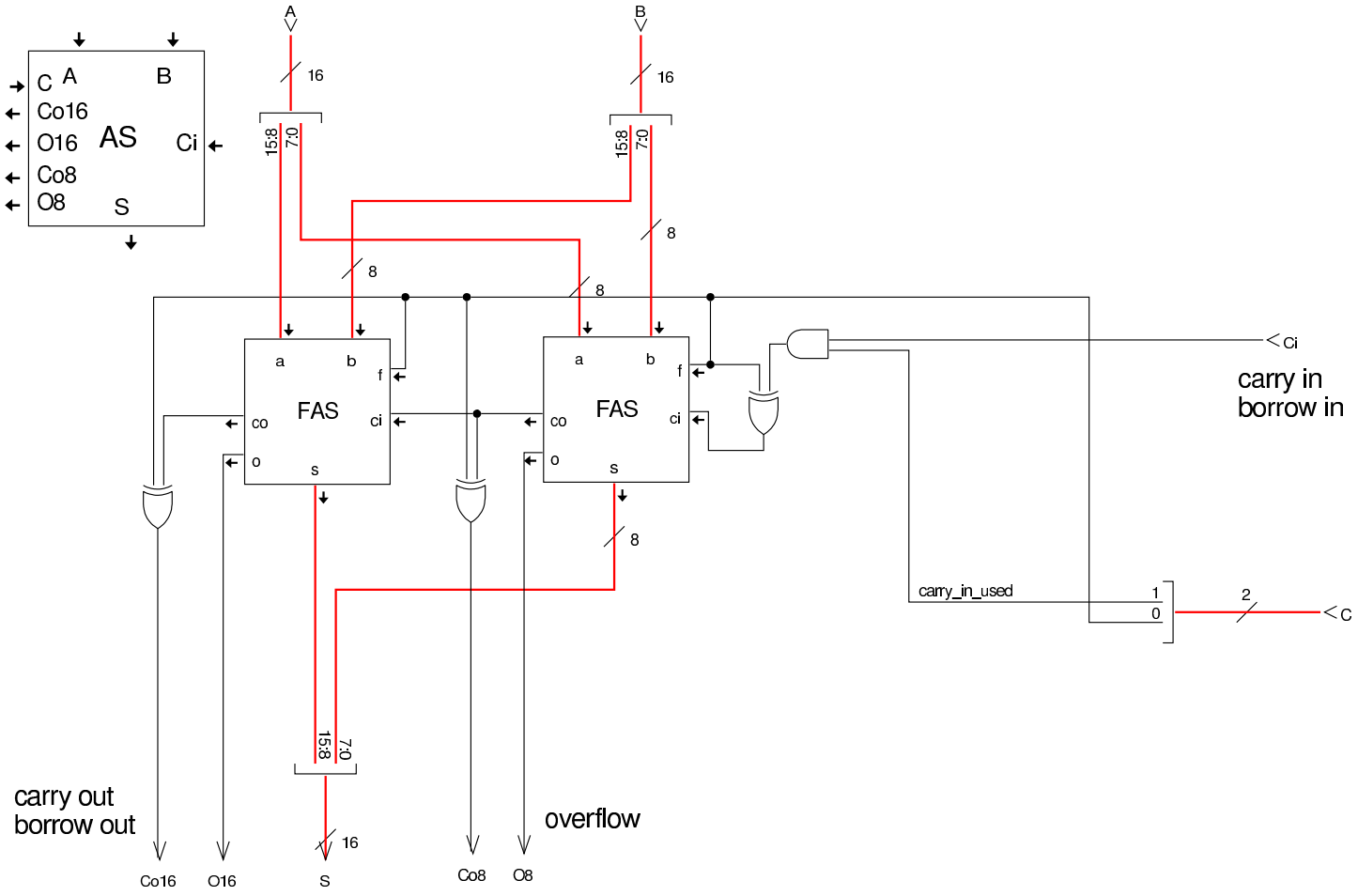


Figure 10.11. Module **`FAS`** (*full adder-subtractor*) that is found inside the arithmetic unit. The **`fa`** are just standard full adders.

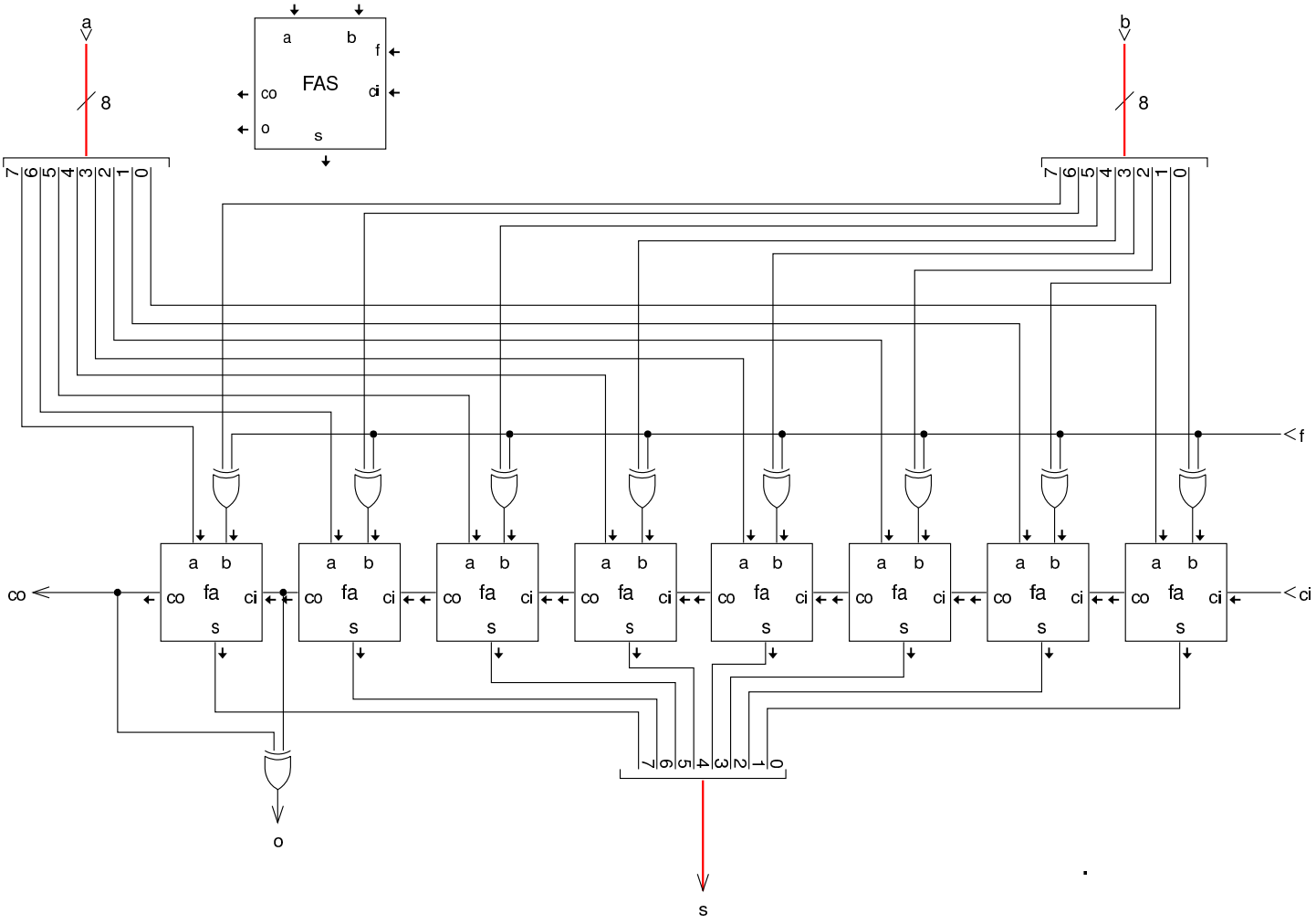


Figure 10.12. Module **SH**, used for bit shift and rotation with carry.

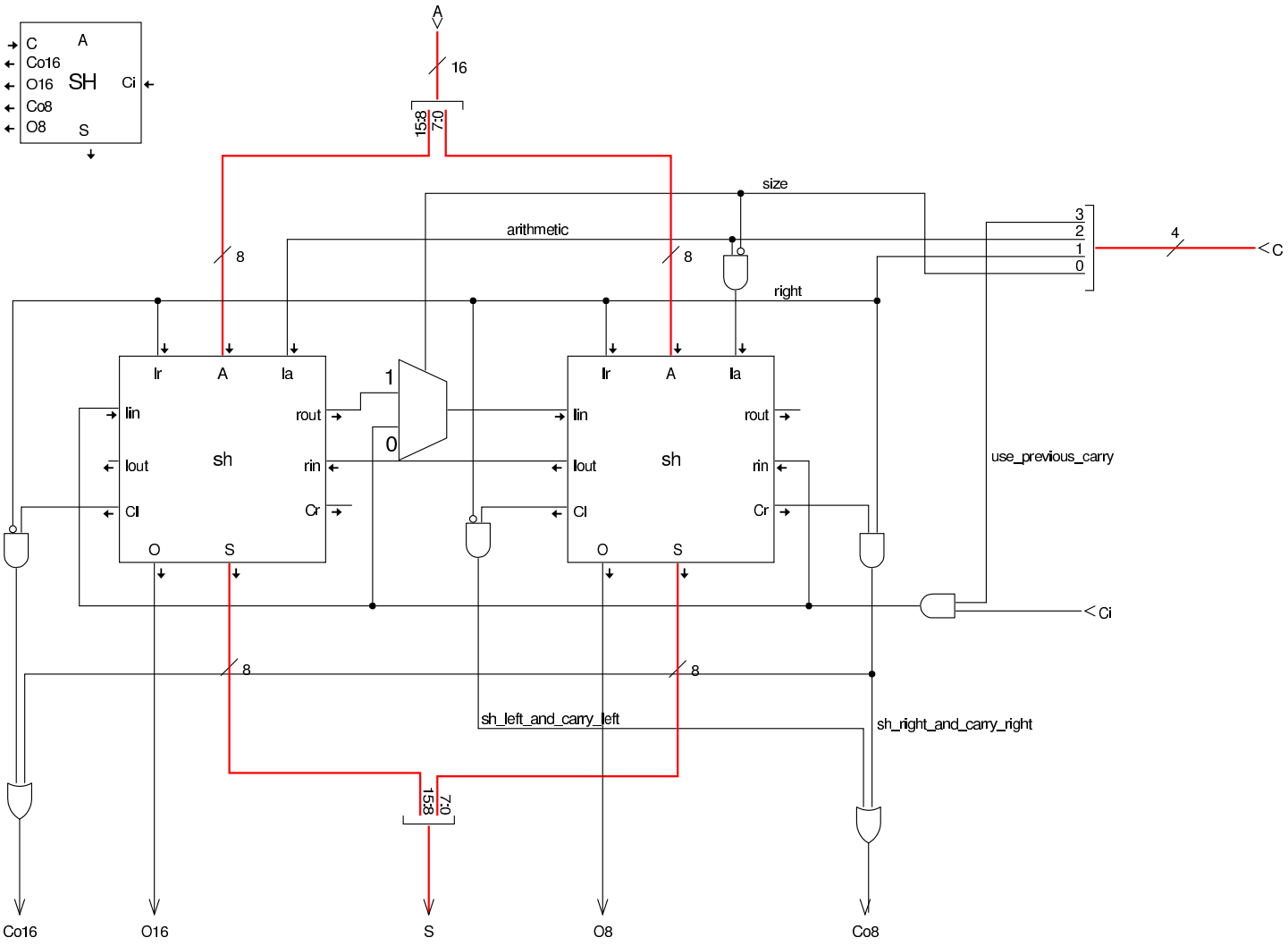


Figure 10.13. Module **ROT**: rotation without carry.

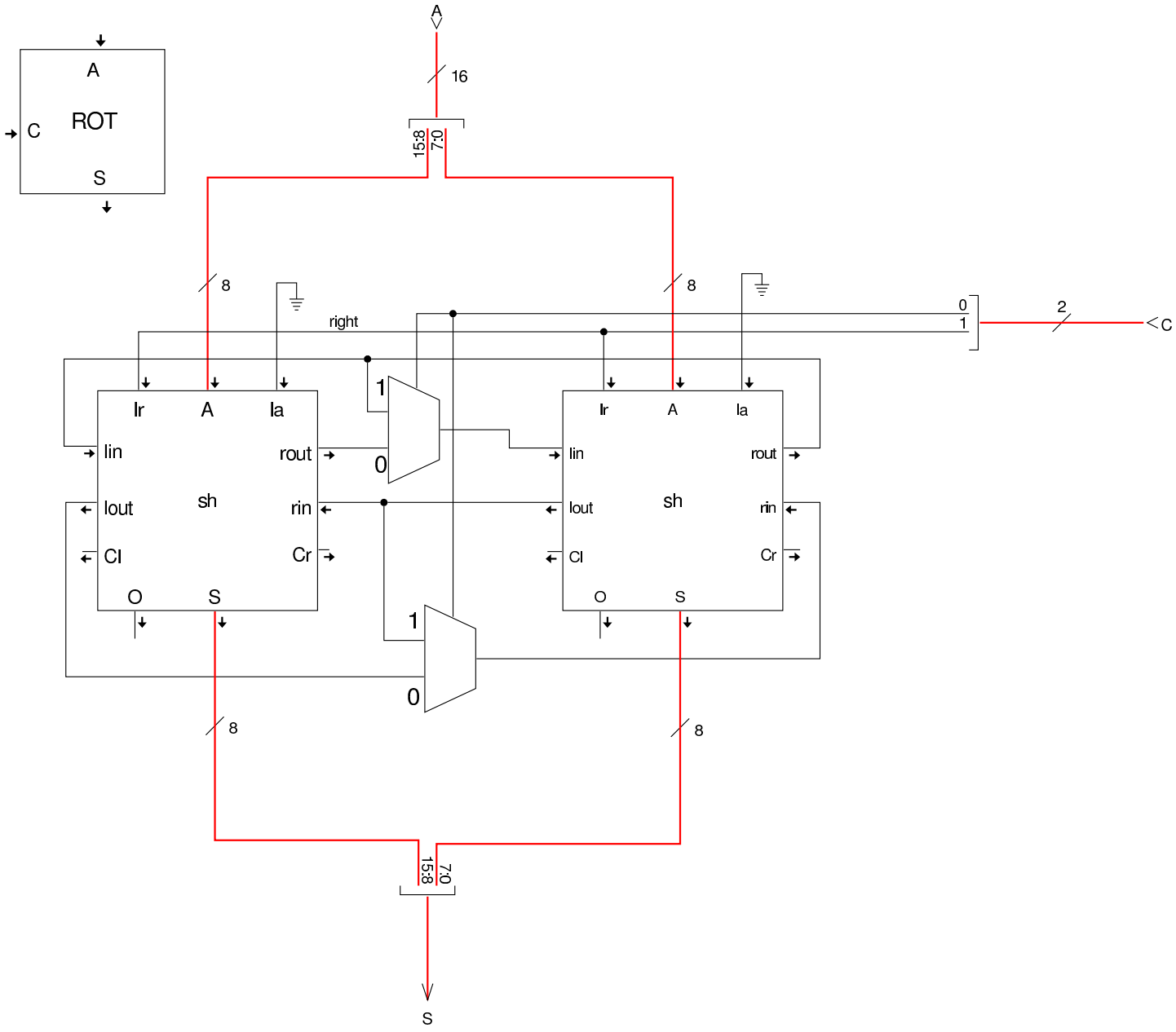
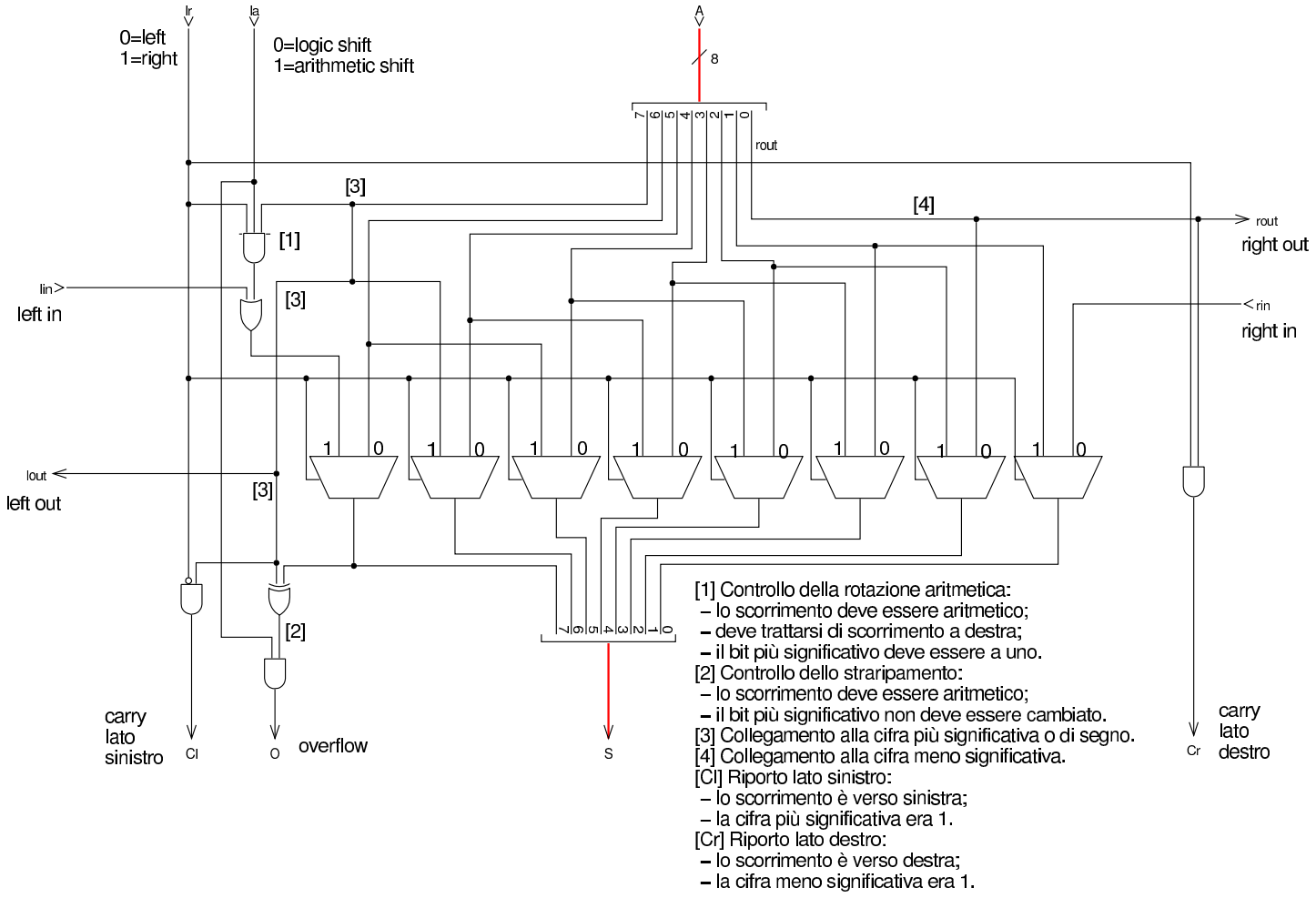
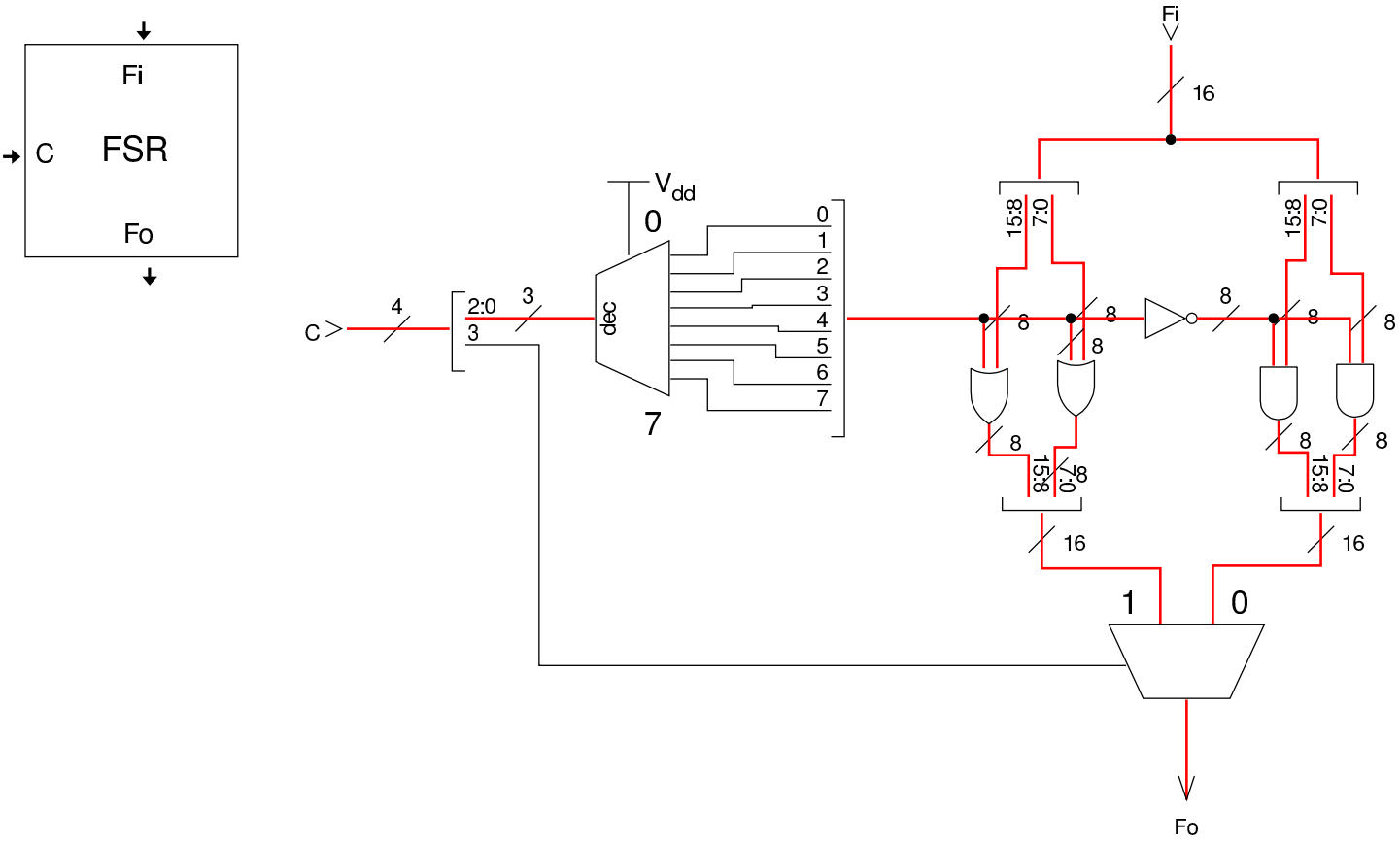


Figure 10.14. Module **sh** used inside the shift and rotation modules.



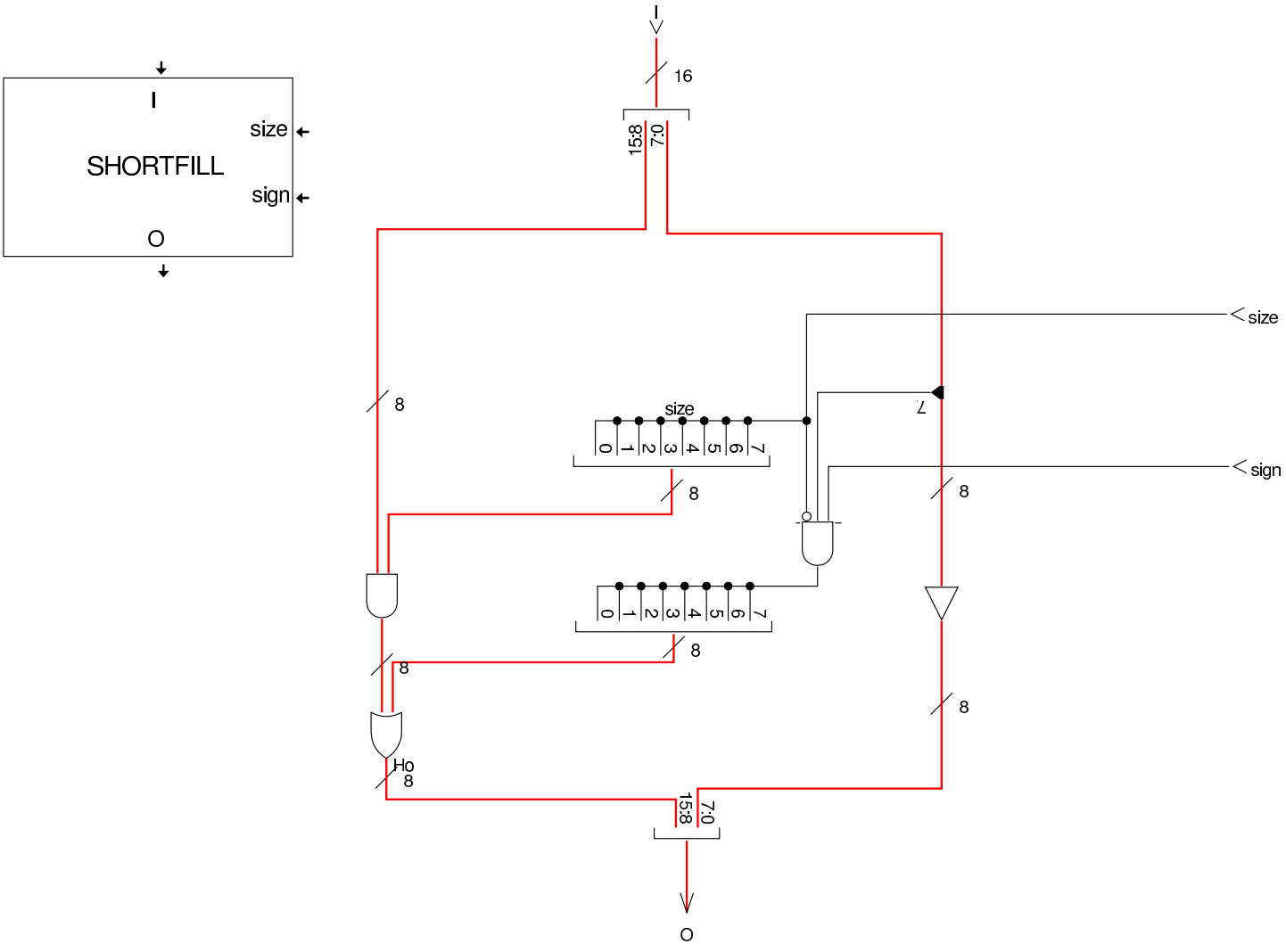
The current ALU is able to modify the flags through the **FAS** module (*flags add-subtract*). As already pointed out, there is a couple of flags, to distinguish between 8-bit and 16-bit results. For example there is a 8-bit carry and another 16-bit carry. When the flags are modified through the **FAS** module, both flags are changed; that is, if the carry flag is changed, this modification is applied to both 8-bit and 16-bit carry flags. The **FAS** module receives a 8-bit mask and the function to be applied (bitwise AND or bitwise OR), to change the flags status.

Figure 10.15. Module **FSR**: direct flags status change.



The module **SHORFILL** is used inside the ALU to reduce the size of a result: if the size reduction is requested, then the most significant byte is filled. If the value is unsigned or positive, the most significant byte is zeroed, otherwise is set to all ones, to extend the sign (if no size reduction is requested, no modification is applied).

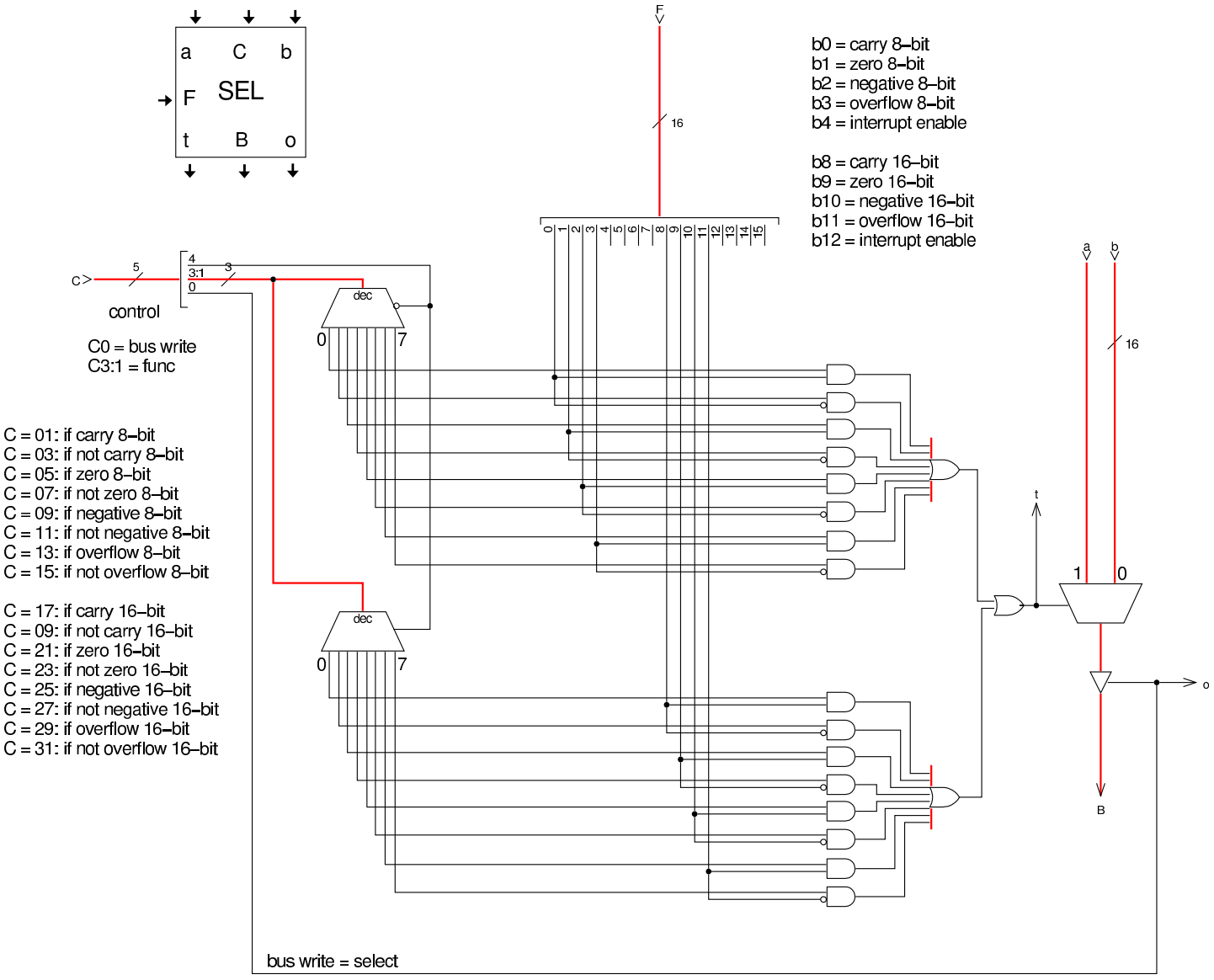
Figure 10.16. Module **SHORTFILL**, for size reduction.



10.4 Module "SEL"

The **SEL** module is extended to handle the distinguished register sets, for eight or sixteen bits. There is also a new flag used to enable hardware interrupts (IRQ), but there are not branch conditions related to the interrupt flag status; that is why the **SEL** module just ignores the interrupt flag.

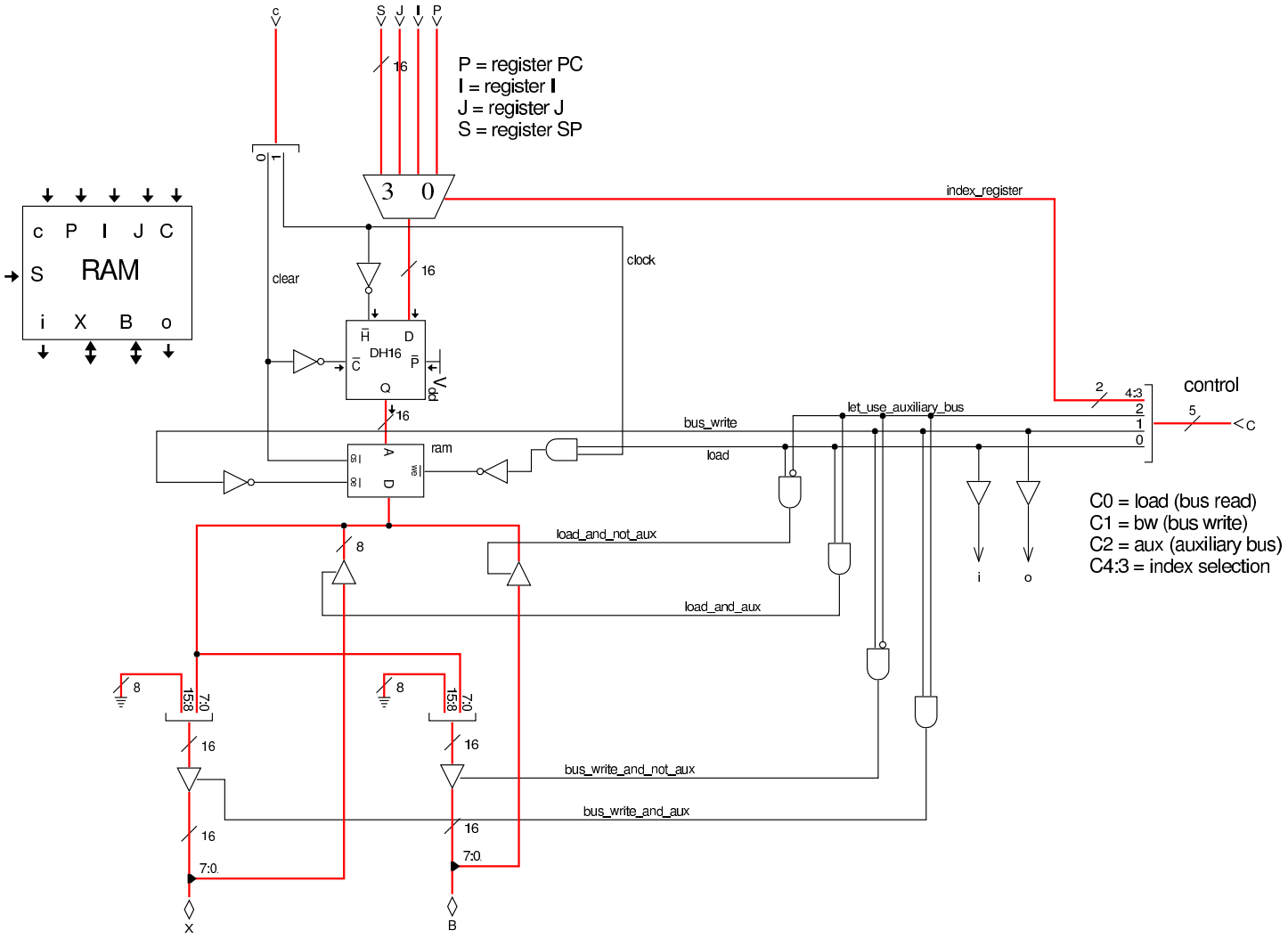
Figure 10.17. Module **SEL**.



10.5 Module "RAM"

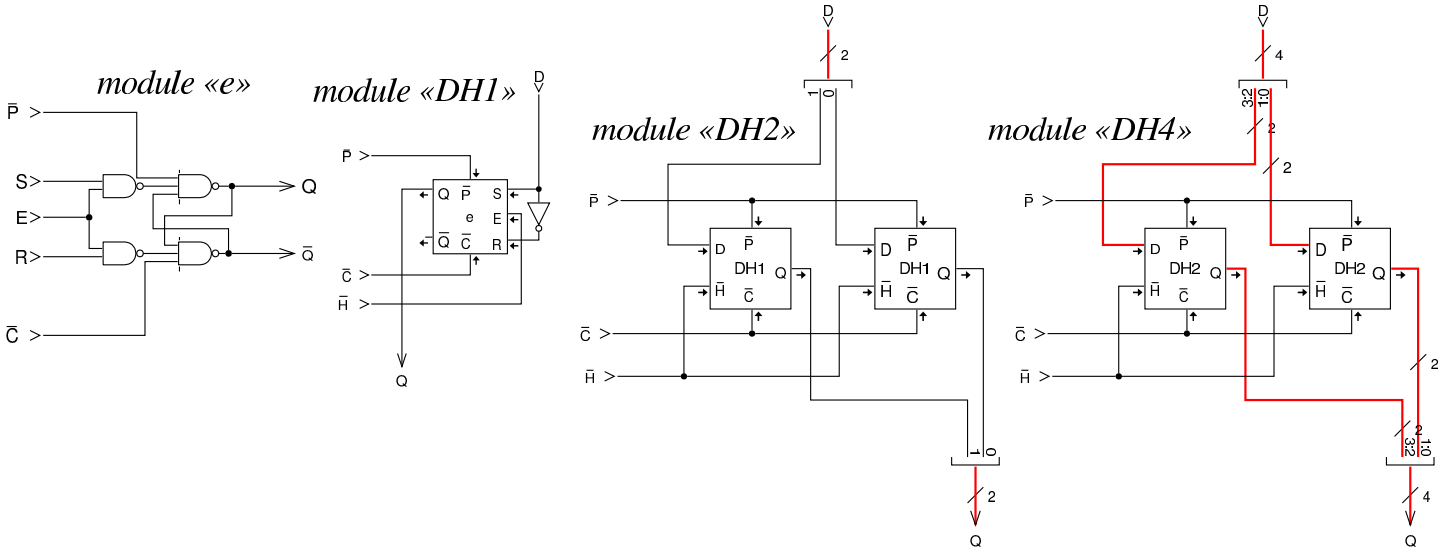
The RAM memory continues to be organized in 8-bit words, as it happens for common architectures. To read or write a 16-bit value, it is necessary to access two times: as the architecture is little-endian, read and write start always from the least significant byte.

Figure 10.18. Module **RAM**.



The **RAM** module contains **DH16** that is a register that repeats on the ***Q*** output the value received form the ***D*** input, as long as the input $\overline{H}$ is negated (active, as the meaning of the line is complemented): when the input line $\overline{H}$ is asserted (hold), the output ***Q*** is freezed.

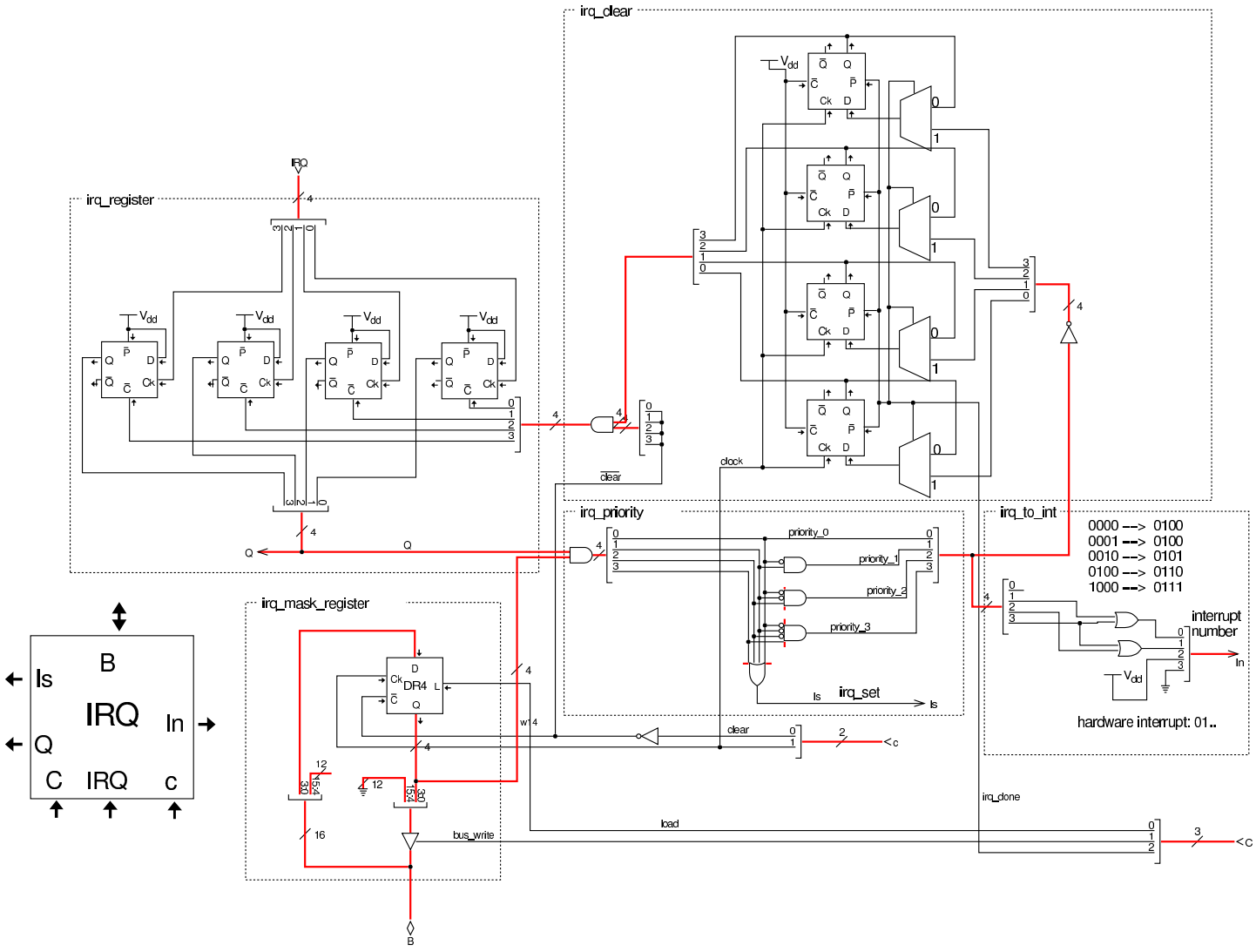
Figure 10.19. From left to right the steps to build **DH*n*** modules: it all starts from a SR latch with enable input; than it becomes a D latch and finally these D latch are put together in paralel. Obviously, the module **DH16** is made with two modules **DH8**, which is made with two modules, **DH4**.



10.6 Module "IRQ"

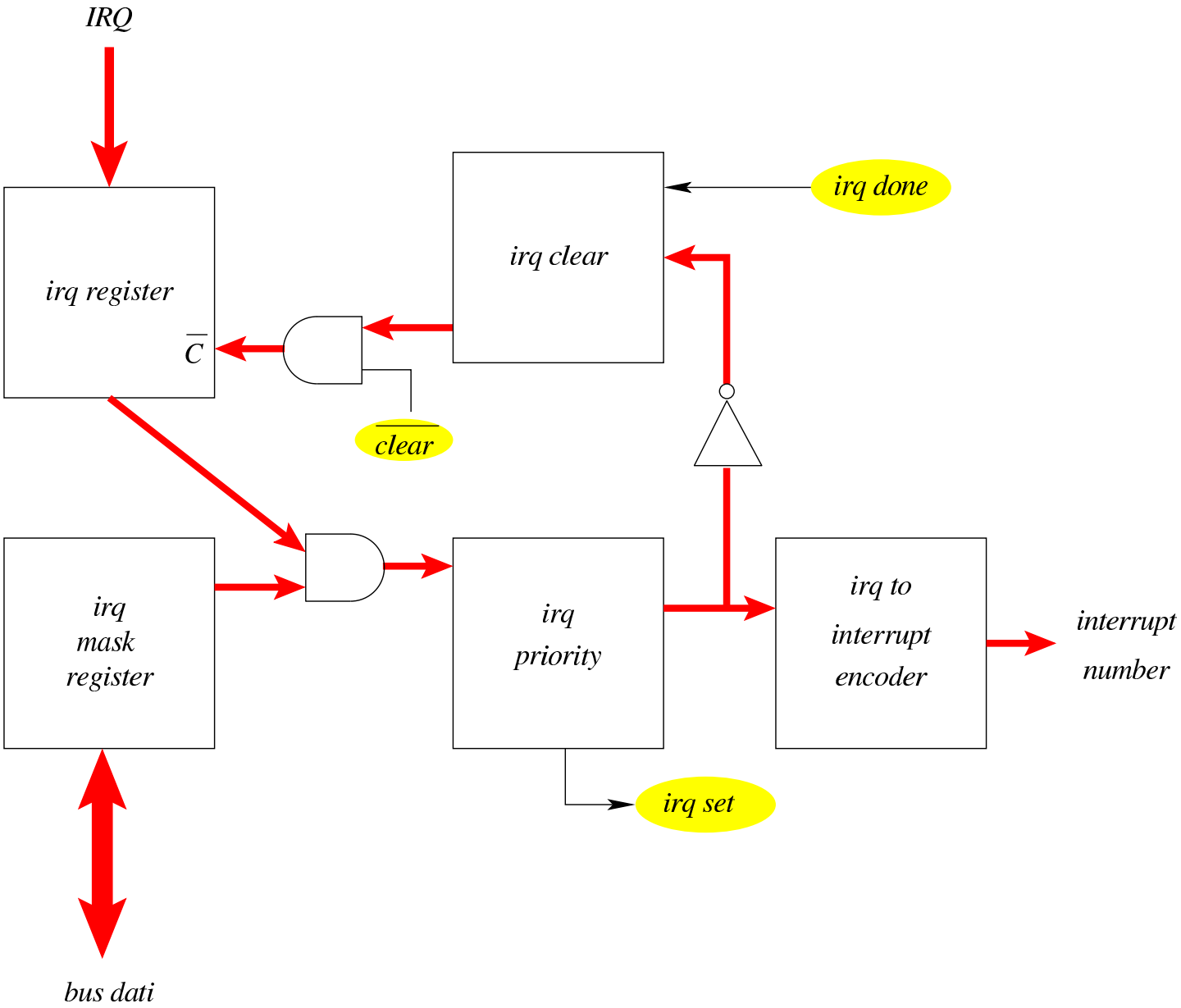
This version of the simple CPU handles interrupts, distinguishing between those produced internally by the CPU itself, those generated by external devices and those managed by software. The **IRQ** module is responsible to collect hardware interrupts from the devices, to sort interrupts by a priority, to supply the selected interrupt to the control circuit that must then do something with it. In short, the module receives IRQ interrupts asynchronously, it stores and determines which is the interruption to be served first. The module appears externally as if it were a register, since it must be able to receive an interrupt mask and the same mask can be read from the module.

Figure 10.20. Module **IRQ**: full schema.



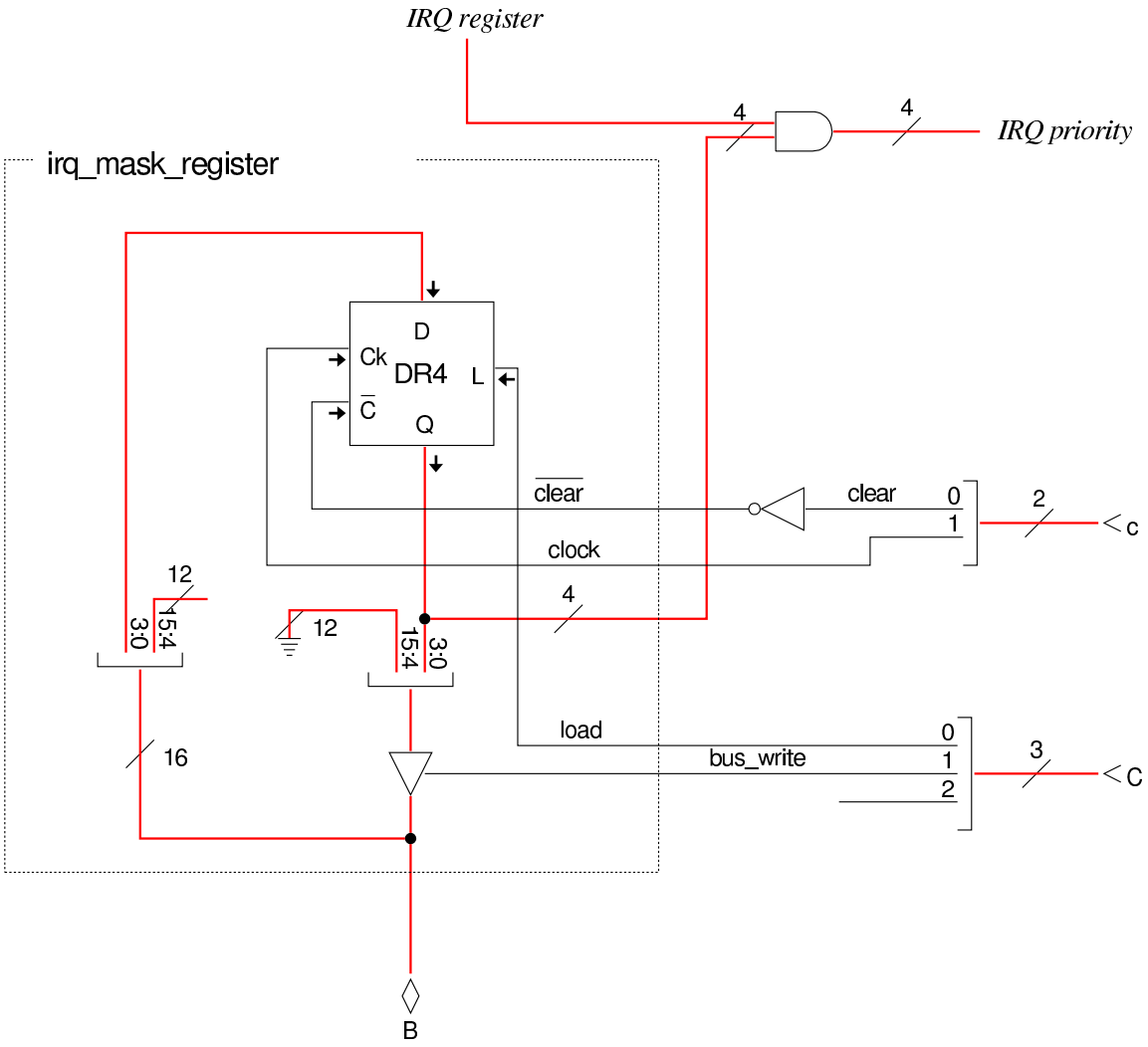
To understand what really does the **IRQ** module, it is necessary to analyze its components, with the help of a block diagram, shown on the next picture.

Figure 10.21. Block diagram for the **IRQ** module.



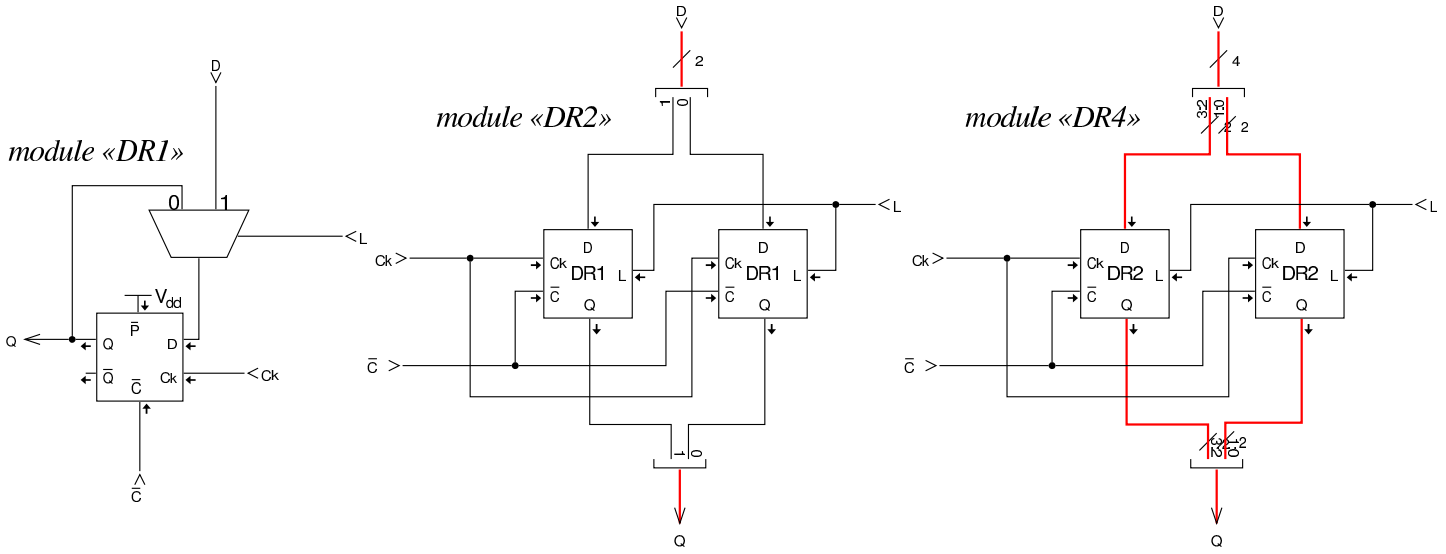
It is simpler to start from the analysis of the register that contains the IRQ mask, which is located at the bottom left of the overall scheme: it is a 4-bit register (one for each IRQ) which reads data from the bus to update its value and writes to the data bus to allow to know the value that it contains.

Figure 10.22. IRQ mask detail.



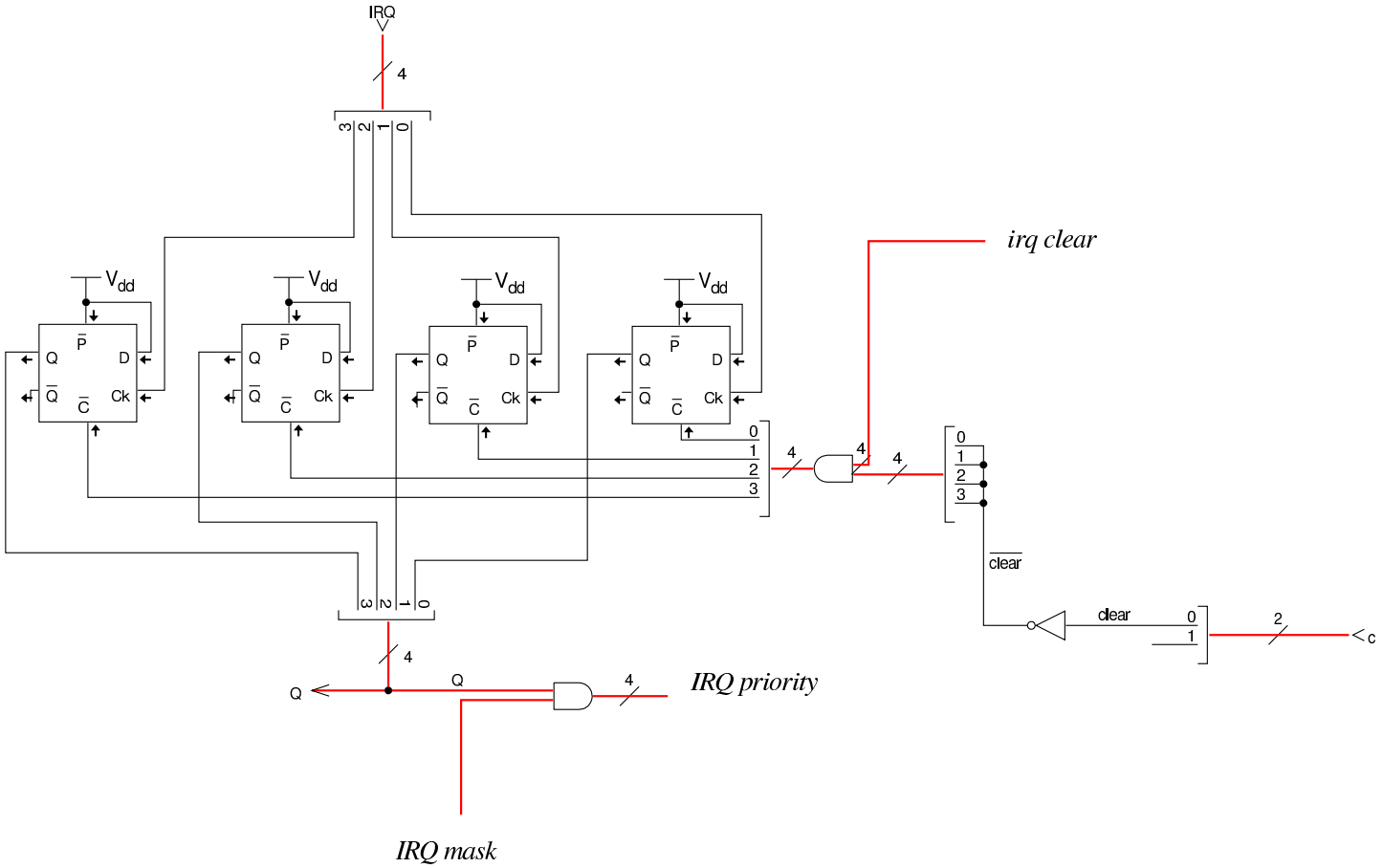
The **DR4** module is a 4-bit register, made of D flip-flops, through the following steps.

Figure 10.23. Building **DR*n*** modules.



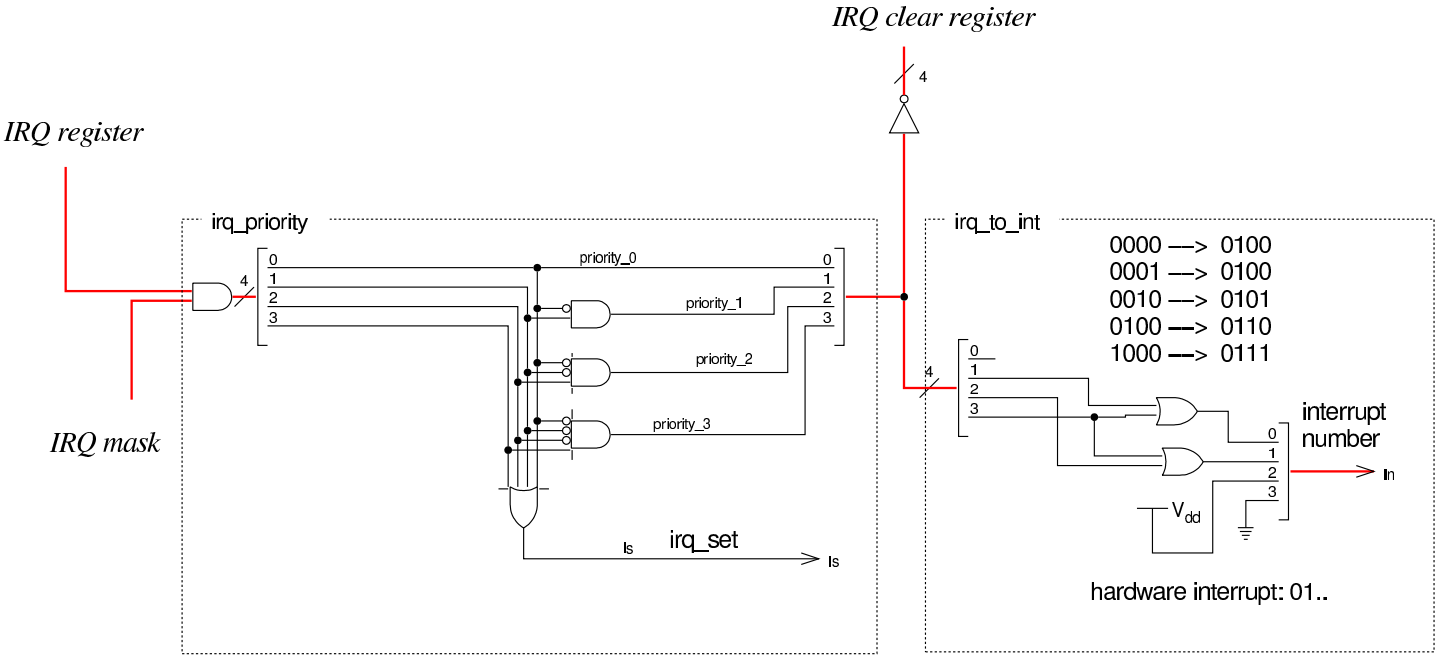
In the upper left of the overall scheme, it appears the IRQ register, whose purpose is to store the hardware interrupts received from the input ***IRQ***. this register is constructed in an unusual way, because it is made by D flip-flops, positive edge, but the ***D*** inputs of these flip-flops is connected so as to be always active, while the clock input is used to receive the IRQ signal. In practice, a IRQ signal arriving at the flip-flop clock line, activates permanently the flip-flop. The IRQ register flip-flops can be reset to zero only asserting the input line $\overline{C}$.

Figure 10.24. Received IRQ register.



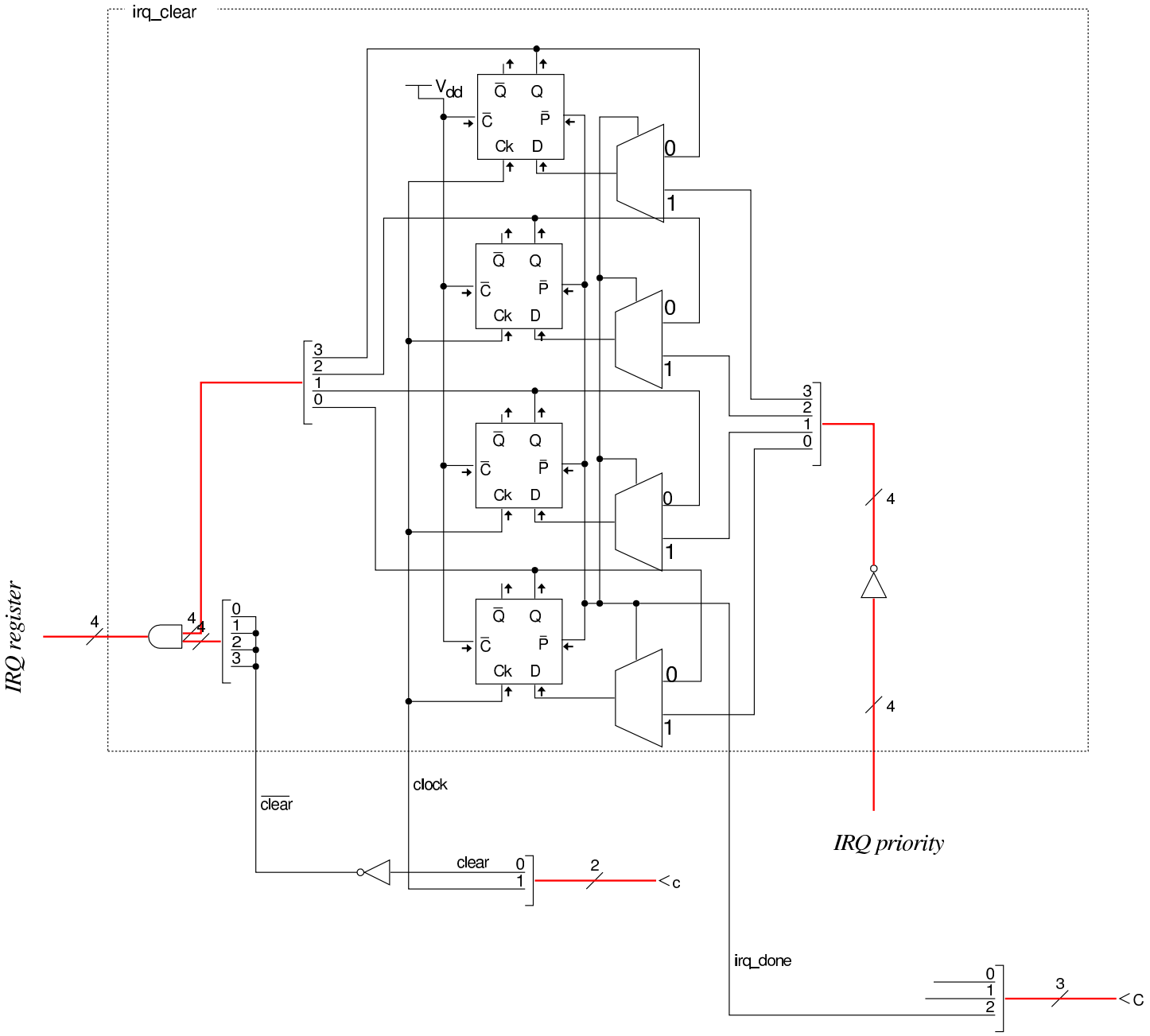
The value stored inside the IRQ register and the value from the IRQ mask below, are compared with a multiple AND gate (a separate gate for each IRQ line) and then given to a module that selects one IRQ line according to a priority: the lowest IRQ number requested is chosen. The module responsible for the priority selection communicates with an encoder that takes care of transforming the IRQ chosen into a number of interruption: IRQ0 becomes INT4, IRQ1 becomes INT5, up to IRQ3 which becomes INT7. It should be observed that the priority selection module emits a signal (***irq_set***) about the actual presence of an IRQ that needs to be served, because the absence of an IRQ produces anyway the value INT4 through the encoder.

Figure 10.25. Priority selection and encoder.



When an IRQ has been served, there is the need to reset the matching D flip-flop in the IRQ register (top left). To obtain this result a reset register is used (it appears at the top right). The reset register is made of D flip-flops (positive edge) which under normal conditions (when the input ***irq_done*** is zero) produce at their output a value equal to one, since they are initialized to one (input $\overline{\boldsymbol{P}}$ to zero). The output of the reset register is connected to the $\overline{\boldsymbol{C}}$ input of the IRQ register: as long as the $\overline{\boldsymbol{C}}$ lines of the IRQ register are negated (in that case it means that the lines are active), the IRQ register retains its stored value. However, when the reset register receives the signal ***irq_done*** it gets and store the complemented IRQ (selected by the priority); that is: the last selected IRQ becomes zero inside the reset register and then, asserts the matching $\overline{\boldsymbol{C}}$ input line of the IRQ register, resetting it. At that point, the priority module might select a new IRQ requesting to be served.

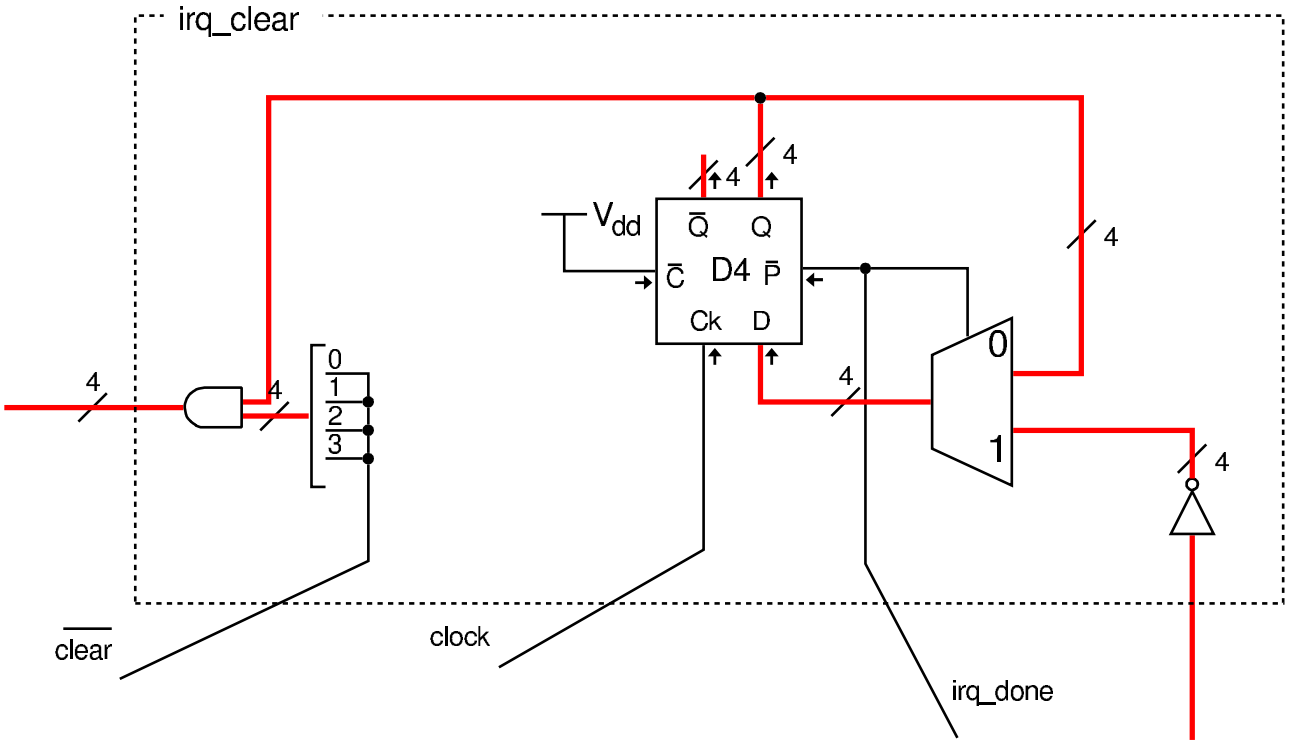
Figure 10.26. Served IRQ reset register.



The $\overline{C}$ inputs at the IRQ register on the left are not connected directly to the served IRQ reset register, because it might be necessary to reset the whole IRQ register, through the control reset line. For this reason, there is an AND gate to handle that situation.

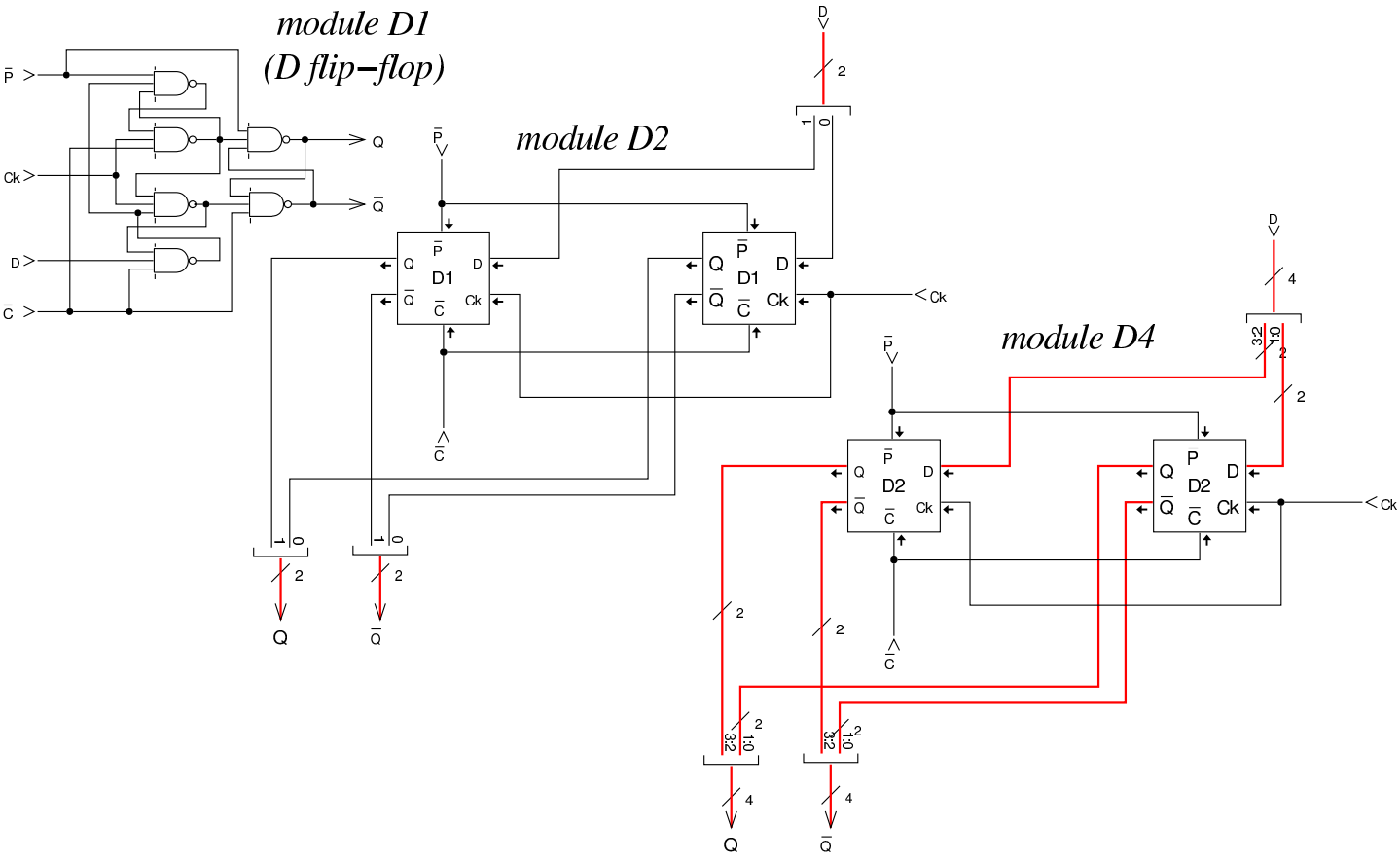
The served IRQ reset register might be designed in a more compact way, as the following picture shows.

Figure 10.27. served IRQ reset register, made of a module **D4**.



The module **D*n*** is made of D flip-flops, in paralel, as the following figure shows.

Figure 10.28. Building **Dn** modules.

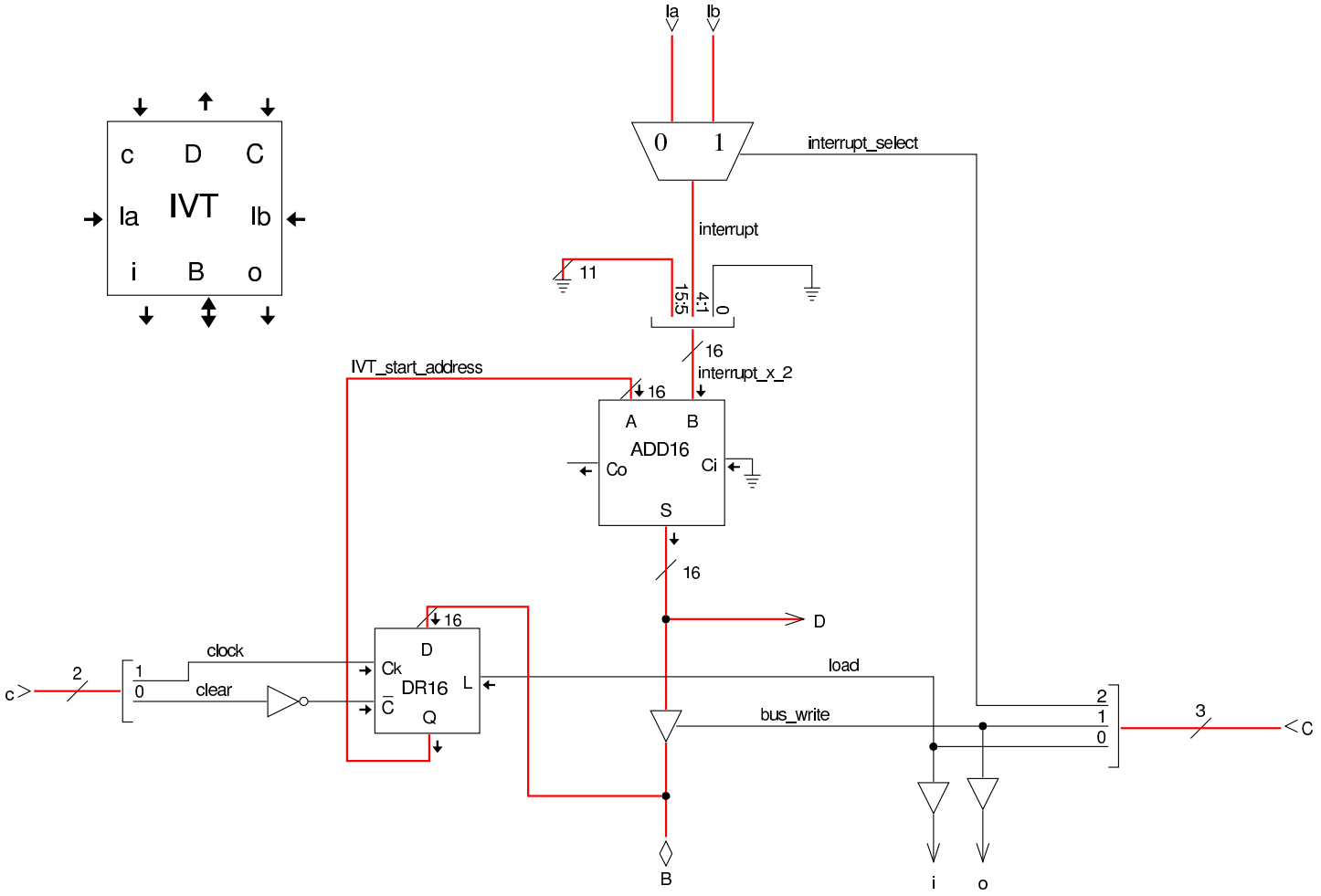


10.7 Module "IVT"

In order to manage interruptions (CPU, hardware and software), this version of the simple CPU needs to have a IVT table (interrupt vector table), which resides in RAM memory. The table IVT must be implemented as an array of 16-bit integers (little-endian), each of which represents the address of a routine to execute when the matching interrupt is enabled. Therefore, ***IVT[n]*** must match the address of the routine that has to perform the activity requested by the interrupt ***n***.

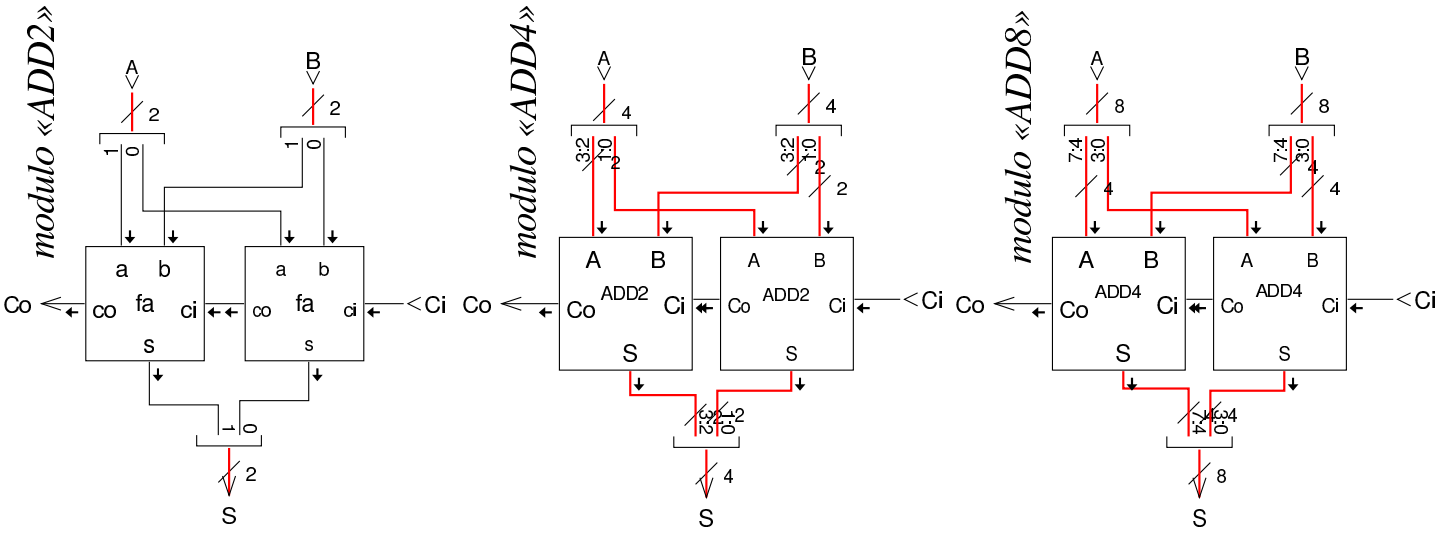
The module **IVT** is used to store the location of the IVT table, which is precisely the location of ***IVT[0]***. The module receives the number of a certain interruption from two independent inputs; such interrupt number is then transformed into the matching address in memory that contains the reference to the requested routine to be started.

Figure 10.29. Module **IVT**.



Inside the **IVT** module there is a register represented by the module **DR16**. The module **DR16** is used to store the IVT table starting address. The module **DR16** is made in the same manner already described for the module **DR4**, inside the previous section. The module **ADD16** is made of sixteen full adders, connected in parallel with a cascade carry. Even the **ADD16** module is made through different stages, as it is for **DR16**.

Figure 10.30. Building modules **ADD*n***.



10.8 Module "CTRL"

The module **CTRL** has only minor changes from the previous version: the opcode occupy only eight bits (input ***I***), the microprogram counter (**MPC**) is reduced to only nine bits, because the microcode does not exceed 512 lines, the microcode word requires many more bits, so, five memory modules are used together to drive a control bus of 160 bits. The input to the microprogram counter is mediated by a multiplexer that permits to enter an address when the D flip-flop on the right is active. This address must match the point where the microcode describes the steps required to serve a hardware interrupt (IRQ); in practice, such address must match the **irq** label placement, as it can be determined from the files produced by the TKGate compiler.

Figure 10.31. Module **CTRL**.

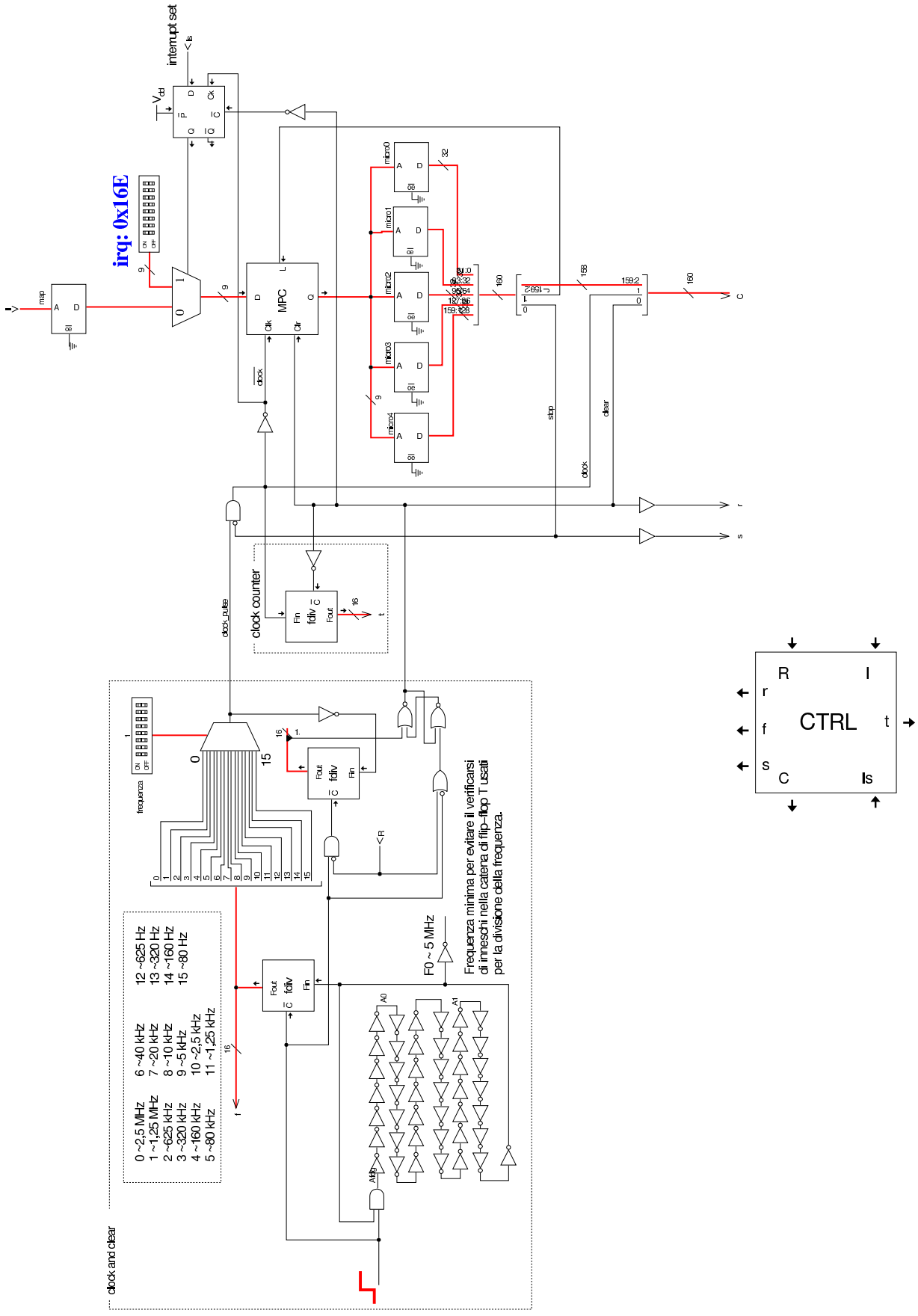
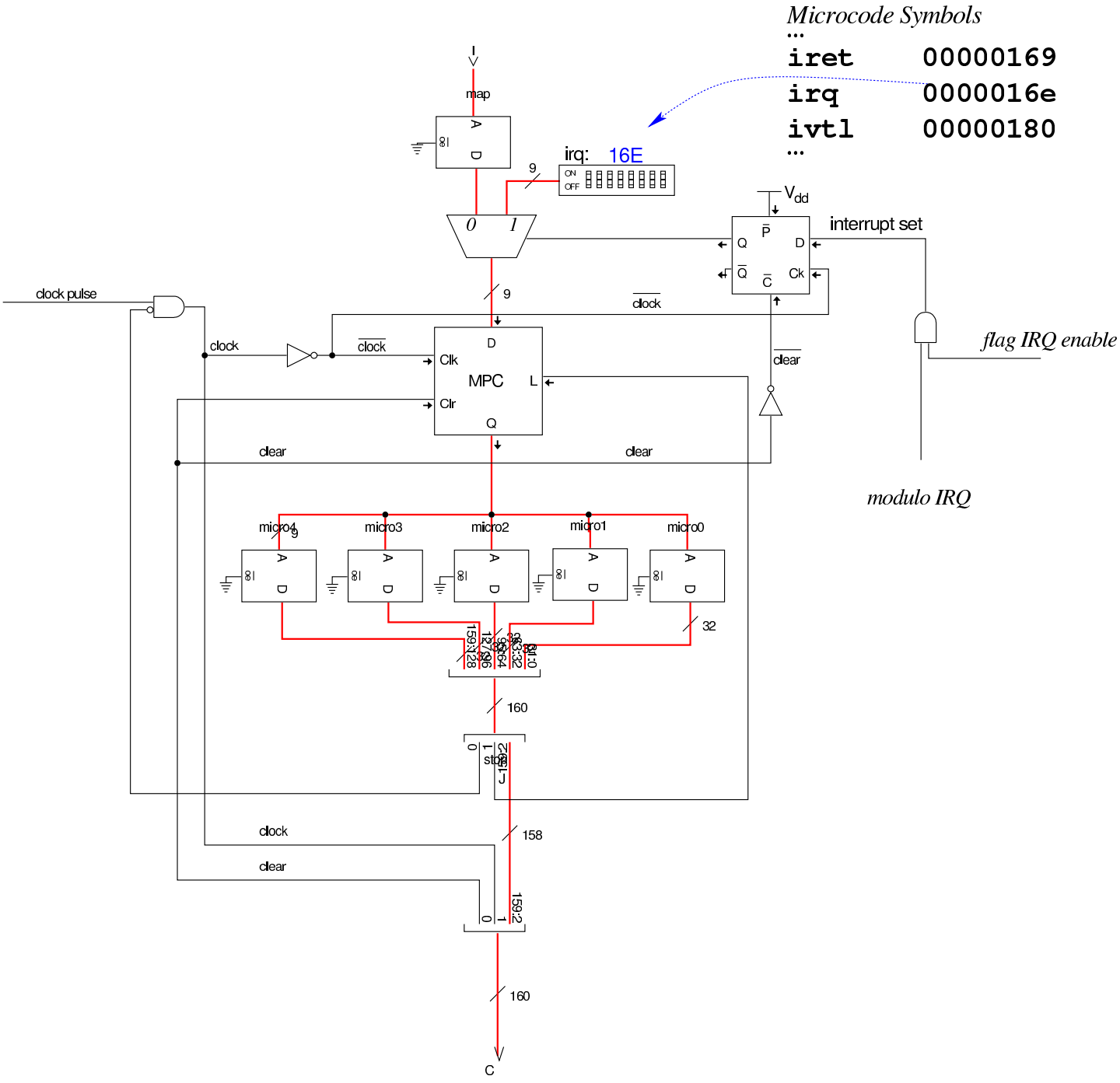


Figure 10.32. Control unit detail.



Looking at the figure above, it is important to clarify what happens when a hardware interrupt is received: the **IRQ** module, as described in section 10.6, sends to the **IVT** module the interrupt number matching the selected IRQ to be served; then, the **IRQ** module activates the ***Is*** output (*interrupt set*). The interrupt-set line activated by the **IRQ** module, reaches the control unit only if the interrupt-enable flag (***I***) is active. When the interrupt request reaches the module **CTRL** at the input ***Is***, it is stored into the D flip-flop (positive edge) that appears in the upper right of the diagram; then,

at the first occasion in which the control unit must read a new opcode, it is instead directed to execute the instructions matching the label **irq**.

Listing 10.33. Memory banks declaration.

```
map        bank[7:0]     ctrl.map;
microcode  bank[31:0]    ctrl.micro0;
microcode  bank[63:32]   ctrl.micro1;
microcode  bank[95:64]   ctrl.micro2;
microcode  bank[127:96]  ctrl.micro3;
microcode  bank[159:128] ctrl.micro4;
macrocode  bank[15:0]    ram.ram;
```

10.9 Opcodes

«

The opcode size used in this version of the simple CPU, remains limited to eight bits, but instead of using just a sequential number to distinguish them, the opcodes are now structured with a criterion. First there is a definition used to convert register names to integer numbers, so that three bits are requested to distinguish them:

```
registers I=0, J=1, A=2, B=3, BP=4, SP=5, MDR=6, FL=7;
```

There are different kind of operands; the simplest cases are declared at the beginning:

```
operands op_0 {
    -  = { };
};
operands op_8 {
    #1 = { +1=#1[7:0]; };
};
operands op_16 {
    #1 = { +1=#1[7:0]; +2=#1[15:8]; };
};
```

It is understood that **op_0** means the request of no operand at all; that **op_8** requires an 8-bit operand; that **op_16** requires a 16-bit operand. For the 16-bit operand it is requested to read two separate bytes, starting from the least significant one, because

the byte order organization is little-endian.

Listing 10.36. Opcodes declaration.

```
op nop {
  map nop:        0x00;  // not operate
  +0[7:0]=0x00;
  operands op_0;
};
op mv {                  // 00...... = mv
  map nop:        0x00;  // 00000000 = mv %I %I not valid => nop
  map mv_i_j:     0x01;  // 00000001 = mv %I %J
  map mv_i_a:     0x02;  // 00000010 = mv %I %A
  map mv_i_b:     0x03;  // 00000011 = mv %I %B
  map mv_i_bp:    0x04;  // 00000100 = mv %I %BP
  map mv_i_sp:    0x05;  // 00000101 = mv %I %SP
  map mv_i_mdr:   0x06;  // 00000110 = mv %I %MDR
  map mv_i_fl:    0x07;  // 00000111 = mv %I %FL
  map mv_j_i:     0x08;  // 00001000 = mv %J %I
  map op_error:   0x09;  // 00001001 = mv %J %J not valid
  map mv_j_a:     0x0A;  // 00001010 = mv %J %A
  map mv_j_b:     0x0B;  // 00001011 = mv %J %B
  map mv_j_bp:    0x0C;  // 00001100 = mv %J %BP
  map mv_j_sp:    0x0D;  // 00001101 = mv %J %SP
  map mv_j_mdr:   0x0E;  // 00001110 = mv %J %MDR
  map mv_j_fl:    0x0F;  // 00001111 = mv %J %FL
  map mv_a_i:     0x10;  // 00010000 = mv %A %I
  map mv_a_j:     0x11;  // 00010001 = mv %A %J
  map op_error:   0x12;  // 00010010 = mv %A %A not valid
  map mv_a_b:     0x13;  // 00010011 = mv %A %B
  map mv_a_bp:    0x14;  // 00010100 = mv %A %BP
  map mv_a_sp:    0x15;  // 00010101 = mv %A %SP
  map mv_a_mdr:   0x16;  // 00010110 = mv %A %MDR
  map mv_a_fl:    0x17;  // 00010111 = mv %A %FL
  map mv_b_i:     0x18;  // 00011000 = mv %B %I
  map mv_b_j:     0x19;  // 00011001 = mv %B %J
  map mv_b_a:     0x1A;  // 00011010 = mv %B %A
  map op_error:   0x1B;  // 00011011 = mv %B %B not valid
```

```
map mv_b_bp:    0x1C;  // 00011100 = mv %B %BP
map mv_b_sp:    0x1D;  // 00011101 = mv %B %SP
map mv_b_mdr:   0x1E;  // 00011110 = mv %B %MDR
map mv_b_fl:    0x1F;  // 00011111 = mv %B %FL
map mv_bp_i:    0x20;  // 00100000 = mv %BP %I
map mv_bp_j:    0x21;  // 00100001 = mv %BP %J
map mv_bp_a:    0x22;  // 00100010 = mv %BP %A
map mv_bp_b:    0x23;  // 00100011 = mv %BP %B
map op_error:   0x24;  // 00100100 = mv %BP %BP not valid
map mv_bp_sp:   0x25;  // 00100101 = mv %BP %SP
map mv_bp_mdr:  0x26;  // 00100110 = mv %BP %MDR
map mv_bp_fl:   0x27;  // 00100111 = mv %BP %FL
map mv_sp_i:    0x28;  // 00101000 = mv %SP %I
map mv_sp_j:    0x29;  // 00101001 = mv %SP %J
map mv_sp_a:    0x2A;  // 00101010 = mv %SP %A
map mv_sp_b:    0x2B;  // 00101011 = mv %SP %B
map mv_sp_bp:   0x2C;  // 00101100 = mv %SP %BP
map op_error:   0x2D;  // 00101101 = mv %SP %SP not valid
map mv_sp_mdr:  0x2E;  // 00101110 = mv %SP %MDR
map mv_sp_fl:   0x2F;  // 00101111 = mv %SP %FL
map mv_mdr_i:   0x30;  // 00110000 = mv %MDR %I
map mv_mdr_j:   0x31;  // 00110001 = mv %MDR %J
map mv_mdr_a:   0x32;  // 00110010 = mv %MDR %A
map mv_mdr_b:   0x33;  // 00110011 = mv %MDR %B
map mv_mdr_bp:  0x34;  // 00110100 = mv %MDR %BP
map mv_mdr_sp:  0x35;  // 00110101 = mv %MDR %SP
map op_error:   0x36;  // 00110110 = mv %MDR %MDR not valid
map mv_mdr_fl:  0x37;  // 00110111 = mv %MDR %FL
map mv_fl_i:    0x38;  // 00111000 = mv %FL %I
map mv_fl_j:    0x39;  // 00111001 = mv %FL %J
map mv_fl_a:    0x3A;  // 00111010 = mv %FL %A
map mv_fl_b:    0x3B;  // 00111011 = mv %FL %B
map mv_fl_bp:   0x3C;  // 00111100 = mv %FL %BP
map mv_fl_sp:   0x3D;  // 00111101 = mv %FL %SP
map mv_fl_mdr:  0x3E;  // 00111110 = mv %FL %MDR
map op_error:   0x3F;  // 00111111 = mv %FL %FL not valid
```

```
  +0[7:0]=0x00;
  operands {
    %1,%2 = { +0[5:3]=%1; +0[2:0]=%2; };
  };
};
op load8 {                  // 010001.. = load8
  map load8_i:    0x44;    // 01000100 = load8 %I
  map load8_j:    0x45;    // 01000101 = load8 %J
  map load8:      0x46;    // 01000110 = load8 #...
  map op_error:   0x47;    // 01000111 = not valid
  +0[7:0]=0x44;
  operands {
    (%1) = { +0[1]=0; +0[0]=%1; };
    #1 = { +0[1]=1; +0[0]=0; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
op load16 {                 // 010010.. = load16
  map load16_i:   0x48;    // 01001000 = load16 %I
  map load16_j:   0x49;    // 01001001 = load16 %J
  map load16:     0x4A;    // 01001010 = load16 #...
  map op_error:   0x4B;    // 01001011 = not valid
  +0[7:0]=0x48;
  operands {
    (%1) = { +0[1]=0; +0[0]=%1; };
    #1 = { +0[1]=1; +0[0]=0; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
op store8 {                 // 010011.. = store8
  map store8_i:   0x4C;    // 01001100 = store8 %I
  map store8_j:   0x4D;    // 01001101 = store8 %J
  map store8:     0x4E;    // 01001110 = store8 #...
  map op_error:   0x4F;    // 01001111 = not valid
  +0[7:0]=0x4C;
  operands {
    (%1) = { +0[1]=0; +0[0]=%1; };
    #1 = { +0[1]=1; +0[0]=0; +1=#1[7:0]; +2=#1[15:8]; };
```

```
  };
};
op store16 {                  // 010100.. = store16
  map store16_i: 0x50;        // 01010000 = store16 %I
  map store16_j: 0x51;        // 01010001 = store16 %J
  map store16:   0x52;        // 01010010 = store16 #...
  map op_error:  0x53;        // 01010011 = not valid
  +0[7:0]=0x50;
  operands {
    (%1) = { +0[1]=0; +0[0]=%1; };
    #1 = { +0[1]=1; +0[0]=0; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
op cp8 {                      // 0101010. = cp8
  map cp8_ij:    0x54;        // 01010100 = cp8 %I
  map cp8_ji:    0x55;        // 01010101 = cp8 %J
  +0[7:0]=0x54;
  operands {
    (%1) = { +0[0]=%1; };
  };
};
op cp16 {                     // 0101011. = cp16
  map cp16_ij:   0x56;        // 01010110 = cp16 %I
  map cp16_ji:   0x57;        // 01010111 = cp16 %J
  +0[7:0]=0x56;
  operands {
    (%1) = { +0[0]=%1; };
  };
};
op return {                   // 010110..
  map return:    0x58;        // 01011000 = return
  +0[7:0]=0x58;
  operands op_0;
};
op call {                     // 010110..
  map call:      0x59;        // 01011001 = call #...
```

```
  map call_i:     0x5A;  // 01011010 = call %I
  map call_j:     0x5B;  // 01011011 = call %J
  +0[7:0]=0x58;
  operands {
    #1 = { +0[1]=0; +0[0]=1; +1=#1[7:0]; +2=#1[15:8]; };
    (%1) = { +0[1]=1; +0[0]=%1; };
  };
};
op int {
  map int:        0x5C;  // 01011100 = int #...
  +0[7:0]=0x5C;
  operands op_8;
};
op iret {
  map iret:       0x5D;  // 01011101 = iret
  +0[7:0]=0x5D;
  operands op_0;
};
op cleari {
  map cleari:     0x5E;  // 01011110 = clear interrupt flag
  +0[7:0]=0x5E;
  operands op_0;
};
op seti {
  map seti:       0x5F;  // 01011111 = set interrupt flag
  +0[7:0]=0x5F;
  operands op_0;
};
op ivtl {
  map ivtl:       0x60;  // 01100000 = load IVT location
  +0[7:0]=0x60;
  operands op_16;
};
op jump {
  map jump:       0x61;  // 01100001 = jump #...
  +0[7:0]=0x61;
```

```
  operands op_16;
};
op jump8c {
  map jump8c:    0x62;  // 01100010 = jump8c #...
  +0[7:0]=0x62;
  operands op_16;
};
op jump8nc {
  map jump8nc:   0x63;  // 01100011 = jump8nc #...
  +0[7:0]=0x63;
  operands op_16;
};
op jump8z {
  map jump8z:    0x64;  // 01100100 = jump8z #...
  +0[7:0]=0x64;
  operands op_16;
};
op jump8nz {
  map jump8nz:   0x65;  // 01100101 = jump8nz #...
  +0[7:0]=0x65;
  operands op_16;
};
op jump8o {
  map jump8o:    0x66;  // 01100110 = jump8o #...
  +0[7:0]=0x66;
  operands op_16;
};
op jump8no {
  map jump8no:   0x67;  // 01100111 = jump8no #...
  +0[7:0]=0x67;
  operands op_16;
};
op jump8n {
  map jump8n:    0x68;  // 01101000 = jump8n #...
  +0[7:0]=0x66;
  operands op_16;
```

```
};
op jump8nn {
  map jump8nn:   0x69;  // 01101001 = jump8nn #...
  map op_error:  0x6A;  // 01101010 = not valid
  map op_error:  0x6B;  // 01101011 = not valid
  map op_error:  0x6C;  // 01101100 = not valid
  map op_error:  0x6D;  // 01101101 = not valid
  map op_error:  0x6E;  // 01101110 = not valid
  map op_error:  0x6F;  // 01101111 = not valid
  map op_error:  0x70;  // 01110000 = not valid
  map op_error:  0x71;  // 01110001 = not valid
  +0[7:0]=0x67;
  operands op_16;
};
op jump16c {
  map jump16c:   0x72;  // 01110010 = jump16c #...
  +0[7:0]=0x72;
  operands op_16;
};
op jump16nc {
  map jump16nc:  0x73;  // 01110011 = jump16nc #...
  +0[7:0]=0x73;
  operands op_16;
};
op jump16z {
  map jump16z:   0x74;  // 01110100 = jump16z #...
  +0[7:0]=0x74;
  operands op_16;
};
op jump16nz {
  map jump16nz:  0x75;  // 01110101 = jump16nz #...
  +0[7:0]=0x75;
  operands op_16;
};
op jump16o {
  map jump16o:   0x76;  // 01110110 = jump16o #...
```

```
  +0[7:0]=0x76;
  operands op_16;
};
op jump16no {
  map jump16no:  0x77;  // 01110111 = jump16no #...
  +0[7:0]=0x77;
  operands op_16;
};
op jump16n {
  map jump16n:   0x78;  // 01111000 = jump16n #...
  +0[7:0]=0x76;
  operands op_16;
};
op jump16nn {
  map jump16nn:  0x79;  // 01111001 = jump16 #...
  map op_error:  0x7A;  // 01111010 = not valid
  map op_error:  0x7B;  // 01111011 = not valid
  map op_error:  0x7C;  // 01111100 = not valid
  map op_error:  0x7D;  // 01111101 = not valid
  map op_error:  0x7E;  // 01111110 = not valid
  map op_error:  0x7F;  // 01111111 = not valid
  +0[7:0]=0x77;
  operands op_16;
};
op push8 {              // 10000... = push8
  map push8_i:   0x80;  // 10000000 = push8 %I
  map push8_j:   0x81;  // 10000001 = push8 %J
  map push8_a:   0x82;  // 10000010 = push8 %A
  map push8_b:   0x83;  // 10000011 = push8 %B
  map push8_bp:  0x84;  // 10000100 = push8 %BP
  map op_error:  0x85;  // 10000101 = push8 %SP not valid
  map push8_mdr: 0x86;  // 10000110 = push8 %MDR
  map push8_fl:  0x87;  // 10000111 = push8 %FL
  +0[7:0]=0x80;
  operands {
    %1 = { +0[2:0]=%1; };
```

```
  };
};
op pop8 {                  // 10001... = pop8
  map pop8_i:     0x88;    // 10001000 = pop8 %I
  map pop8_j:     0x89;    // 10001001 = pop8 %J
  map pop8_a:     0x8A;    // 10001010 = pop8 %A
  map pop8_b:     0x8B;    // 10001011 = pop8 %B
  map pop8_bp:    0x8C;    // 10001100 = pop8 %BP
  map op_error:   0x8D;    // 10001101 = pop8 %SP not valid
  map pop8_mdr:   0x8E;    // 10001110 = pop8 %MDR
  map pop8_fl:    0x8F;    // 10001111 = pop8 %FL
  +0[7:0]=0x88;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op push16 {                // 10010... = push16
  map push16_i:   0x90;    // 10010000 = push16 %I
  map push16_j:   0x91;    // 10010001 = push16 %J
  map push16_a:   0x92;    // 10010010 = push16 %A
  map push16_b:   0x93;    // 10010011 = push16 %B
  map push16_bp:  0x94;    // 10010100 = push16 %BP
  map op_error:   0x95;    // 10010101 = push16 %SP not valid
  map push16_mdr: 0x96;    // 10010110 = push16 %MDR
  map push16_fl:  0x97;    // 10010111 = push16 %FL
  +0[7:0]=0x90;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op pop16 {                 // 10011... = pop16
  map pop16_i:    0x98;    // 10011000 = pop16 %I
  map pop16_j:    0x99;    // 10011001 = pop16 %J
  map pop16_a:    0x9A;    // 10011010 = pop16 %A
  map pop16_b:    0x9B;    // 10011011 = pop16 %B
  map pop16_bp:   0x9C;    // 10011100 = pop16 %BP
```

```
  map op_error:  0x9D;  // 10011101 = pop16 %SP not valid
  map pop16_mdr: 0x9E;  // 10011110 = pop16 %MDR
  map pop16_fl:  0x9F;  // 10011111 = pop16 %FL
  +0[7:0]=0x98;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op c8to16u {
  map c8to16u:   0xA0;  // 10100000
  +0[7:0]=0xA0;
  operands op_0;
};
op c8to16s {
  map c8to16s:   0xA1;  // 10100001
  +0[7:0]=0xA1;
  operands op_0;
};
op equal {
  map equal:     0xA2;  // 10100010
  +0[7:0]=0xA2;
  operands op_0;
};
op not {
  map not:       0xA3;  // 10100011
  +0[7:0]=0xA3;
  operands op_0;
};
op and {
  map and:       0xA4;  // 10100100
  +0[7:0]=0xA4;
  operands op_0;
};
op nand {
  map nand:      0xA5;  // 10100101
  +0[7:0]=0xA5;
```

```
  operands op_0;
};
op or {
  map or:         0xA6;  // 10100110
  +0[7:0]=0xA6;
  operands op_0;
};
op nor {
  map nor:        0xA7;  // 10100111
  +0[7:0]=0xA7;
  operands op_0;
};
op xor {
  map xor:        0xA8;  // 10101000
  +0[7:0]=0xA8;
  operands op_0;
};
op nxor {
  map nxor:       0xA9;  // 10101001
  +0[7:0]=0xA9;
  operands op_0;
};
op add {
  map add:        0xAA;  // 10101010
  +0[7:0]=0xAA;
  operands op_0;
};
op sub {
  map sub:        0xAB;  // 10101011
  +0[7:0]=0xAB;
  operands op_0;
};
op addc8 {
  map addc8:      0xAC;  // 10101100
  +0[7:0]=0xAC;
  operands op_0;
```

```
};
op subb8 {
  map subb8:      0xAD;  // 10101101
  +0[7:0]=0xAD;
  operands op_0;
};
op addc16 {
  map addc16:     0xAE;  // 10101110
  +0[7:0]=0xAE;
  operands op_0;
};
op subb16 {
  map subb16:     0xAF;  // 10101111
  +0[7:0]=0xAF;
  operands op_0;
};
op lshl8 {
  map lshl8:      0xB0;  // 10110000
  +0[7:0]=0xB0;
  operands op_0;
};
op lshr8 {
  map lshr8:      0xB1;  // 10110001
  +0[7:0]=0xB1;
  operands op_0;
};
op ashl8 {
  map ashl8:      0xB2;  // 10110010
  +0[7:0]=0xB2;
  operands op_0;
};
op ashr8 {
  map ashr8:      0xB3;  // 10110011
  +0[7:0]=0xB3;
  operands op_0;
};
```

```
op rotcl8 {
  map rotcl8:    0xB4;  // 10110100
  +0[7:0]=0xB4;
  operands op_0;
};
op rotcr8 {
  map rotcr8:    0xB5;  // 10110101
  +0[7:0]=0xB5;
  operands op_0;
};
op rotl8 {
  map rotl8:     0xB6;  // 10110110
  +0[7:0]=0xB6;
  operands op_0;
};
op rotr8 {
  map rotr8:     0xB7;  // 10110111
  +0[7:0]=0xB7;
  operands op_0;
};
op lshl16 {
  map lshl16:    0xB8;  // 10111000
  +0[7:0]=0xB8;
  operands op_0;
};
op lshr16 {
  map lshr16:    0xB9;  // 10111001
  +0[7:0]=0xB9;
  operands op_0;
};
op ashl16 {
  map ashl16:    0xBA;  // 10111010
  +0[7:0]=0xBA;
  operands op_0;
};
op ashr16 {
```

```
  map ashr16:     0xBB;  // 10111011
  +0[7:0]=0xBB;
  operands op_0;
};
op rotcl16 {
  map rotcl16:    0xBC;  // 10111100
  +0[7:0]=0xBC;
  operands op_0;
};
op rotcr16 {
  map rotcr16:    0xBD;  // 10111101
  +0[7:0]=0xBD;
  operands op_0;
};
op rotl16 {
  map rotl16:     0xBE;  // 10111110
  +0[7:0]=0xBE;
  operands op_0;
};
op rotr16 {
  map rotr16:     0xBF;  // 10111111
  +0[7:0]=0xBF;
  operands op_0;
};
op in {
  map in:         0xC0;  // 11000000
  +0[7:0]=0xC0;
  operands op_8;
};
op out {
  map out:        0xC1;  // 11000001
  +0[7:0]=0xC1;
  operands op_8;
};
op ifack_jump {
  map ifack_jump: 0xC2;  // 11000010
```

```
  map op_error:   0xC3;  // 11000011
  +0[7:0]=0xC2;
  operands {
    #1,#2 = { +1=#1[7:0]; +2=#1[7:0]; +3=#2[17:8]; };
  };
};
op clearc {
  map clearc:    0xC4;  // 11000100
  +0[7:0]=0xC4;
  operands op_0;
};
op setc {
  map setc:      0xC5;  // 11000101
  +0[7:0]=0xC5;
  operands op_0;
};
op cmp {
  map cmp:       0xC6;  // 11000110
  +0[7:0]=0xC6;
  operands op_0;
};
op test {
  map test:      0xC7;  // 11000111
  +0[7:0]=0xC7;
  operands op_0;
};
op imrl {
  map imrl:      0xC8;  // 11001000     // IMR load
  map op_error:  0xC9;  // 11001001
  map op_error:  0xCA;  // 11001010
  map op_error:  0xCB;  // 11001011
  map op_error:  0xCC;  // 11001100
  map op_error:  0xCD;  // 11001101
  map op_error:  0xCE;  // 11001110
  map op_error:  0xCF;  // 11001111
  +0[7:0]=0xC8;
```

```
  operands op_8;
};
op inc {
  map inc_i:      0xD0;  // 11010000      // inc %I
  map inc_j:      0xD1;  // 11010001      // inc %J
  map inc_a:      0xD2;  // 11010010      // inc %A
  map inc_b:      0xD3;  // 11010011      // inc %B
  map inc_bp:     0xD4;  // 11010100      // inc %BP
  map inc_sp:     0xD5;  // 11010101      // inc %SP
  map inc_mdr:    0xD6;  // 11010110      // inc %MDR
  map inc_fl:     0xD7;  // 11010111      // inc %FL
  +0[7:0]=0xD0;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op dec {
  map dec_i:      0xD8;  // 11011000      // dec %I
  map dec_j:      0xD9;  // 11011001      // dec %J
  map dec_a:      0xDA;  // 11011010      // dec %A
  map dec_b:      0xDB;  // 11011011      // dec %B
  map dec_bp:     0xDC;  // 11011100      // dec %BP
  map dec_sp:     0xDD;  // 11011101      // dec %SP
  map dec_mdr:    0xDE;  // 11011110      // dec %MDR
  map dec_fl:     0xDF;  // 11011111      // dec %FL
  map op_error:   0xE0;  // 11100001
  map op_error:   0xE1;  // 11100010
  map op_error:   0xE2;  // 11100011
  map op_error:   0xE4;  // 11100100
  map op_error:   0xE5;  // 11100101
  map op_error:   0xE6;  // 11100110
  map op_error:   0xE7;  // 11100111
  map op_error:   0xE8;  // 11101000
  map op_error:   0xE9;  // 11101001
  map op_error:   0xEA;  // 11101010
  map op_error:   0xEB;  // 11101011
```

```
  map op_error:  0xEC;  // 11101100
  map op_error:  0xED;  // 11101101
  map op_error:  0xEE;  // 11101110
  map op_error:  0xEF;  // 11101111
  map op_error:  0xF0;  // 11110001
  map op_error:  0xF1;  // 11110010
  map op_error:  0xF2;  // 11110011
  map op_error:  0xF4;  // 11110100
  map op_error:  0xF5;  // 11110101
  map op_error:  0xF6;  // 11110110
  map op_error:  0xF7;  // 11110111
  map op_error:  0xF8;  // 11111000
  map op_error:  0xF9;  // 11111001
  map op_error:  0xFA;  // 11111010
  map op_error:  0xFB;  // 11111011
  map op_error:  0xFC;  // 11111100
  map op_error:  0xFD;  // 11111101
  map op_error:  0xFE;  // 11111110
  +0[7:0]=0xD8;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op stop {
  map stop:      0xFF;  // 11111111
  +0[7:0]=0xFF;
  operands op_0;
};
```

Table 10.37. Macrocode syntax.

Macrocode syntax	Binary opcode	Description
`nop`	00000000	Not operate: does not to anything.

Macrocode syntax	Binary opcode	Description
`mv %`***src***`, %`***dst***	`00`***sss ddd***	Copy the ***src*** register content inside the ***dst*** register. Allowed registers are: ***I***, ***J***, ***A***, ***B***, ***BP***, ***SP***, ***MDR***, ***FL***. The opcode 00000000_2 remains for the **`nop`** instruction, because it would represent the copy of register ***A*** into the same register ***A***.

`load8 (%I|%J)`
`load8` ***address***
`load16 (%I|%J)`
`load16` ***address***
	`0100010`***i***
	`01000110`
	`0100100`***i***
	`01001010`
		Load inside the ***MDR*** register an 8-bit or 16-bit value from the memory. The argument might be an index register (***I*** or ***J***) and in that case the bit ***i*** is used to distinguish the register inside the opcode; otherwise, the argument can be directly the memory address and the last two bits of the opcode are equal to 10_2.

`store8 (%I|%J)`
`store8` ***address***
`store16 (%I|%J)`
`load16` ***address***
	`0100110`***i***
	`01001110`
	`0101000`***i***
	`01010010`
		Store in memory (eight or sixteen bits), at the address specified by the argument, the value contained inside the ***MDR*** register. If the argument is an index register (***I*** or ***J***), then the bit ***i*** is used to distinguish the register inside the opcode; otherwise, the argument can be directly the memory address and the last two bits of the opcode are equal to 10_2.

`cp8 (%I|%J)` `cp16 (%I|%J)`	`0101010`***s*** `0101011`***s***	Copy from RAM memory to RAM memory eight or sixteen bits, starting from the position represented by the content of the index register specified, to the position represented by the content of the other index register.

Macrocode syntax	Binary opcode	Description

`return`
 	
`call #`***ref***
 	
`call (%I|%J)`
	`01011000`
	
	`01011001`
	
	`0101101`***i***
		Return from ad call to a routine. The bit ***i*** represents an index register (***I*** o ***J***). The **`call`** instruction push the register ***PC*** onto the stack, while the **`return`** instruction restore the register ***PC*** from the stack.

`int` ***n_interrupt***
 	
`iret`
	`01011100`
	
	`01011101`
		Interrupt call and interrupt return; the size of the argument ***n_interrupt*** is 8-bit, but can be only a value from zero to sixteen. When the **`int`** instruction is encountered, the registers ***FL*** and ***PC*** are saved inside the stack and then the IRQ enable flag is zeroed; when an **`iret`** instruction is encountered, the registers ***PC*** and ***FL*** are restored from the stack.

`cleari`
 	
`seti`
	`01011110`
	
	`01011111`
		Reset or set the hardware interrupts flags, contained inside the ***FL*** register.

`ivtl #`***ref***	`01100000`	Load the IVT: interrupt vector table. The argument is the interrupt vector table start address in memory.
`jump #`***ref***	`01100001`	Jump to the address specified.

Macrocode syntax	Binary opcode	Description
`jump8c #`***ref***	`01100010`	Jump to the instruction located at the specified address if: the 8-bit carry flag is active, the 8-bit carry flag is not active, the 8-bit zero flag is active, the 8-bit zero flag is not active, the 8-bit overflow flag is active, the 8-bit overflow flag is not active, the 8-bit sign flag is active, the 8-bit sign flag is not active.
`jump8nc #`***ref***	`01100011`	
`jump8z #`***ref***	`01100100`	
`jump8nz #`***ref***	`01100101`	
`jump8o #`***ref***	`01100110`	
`jump8no #`***ref***	`01100111`	
`jump8n #`***ref***	`01101000`	
`jump8nn #`***ref***	`01101001`	
`jump16c #`***ref***	`01110010`	Jump to the instruction located at the specified address if: the 16-bit carry flag is active, the 16-bit carry flag is not active, the 16-bit zero flag is active, the 16-bit zero flag is not active, the 16-bit overflow flag is active, the 16-bit overflow flag is not active, the 16-bit sign flag is active, the 16-bit sign flag is not active.
`jump16nc #`***ref***	`01110011`	
`jump16z #`***ref***	`01110100`	
`jump16nz #`***ref***	`01110101`	
`jump16o #`***ref***	`01110110`	
`jump16no #`***ref***	`01110111`	
`jump16n #`***ref***	`01111000`	
`jump16nn #`***ref***	`01111001`	

Macrocode syntax	Binary opcode	Description
`push8 %`***register***	`10000`***rrr***	Push on top of the stack the 8-bit content of the specified register, otherwise restore the specified register removing the last 8-bit value from the top of the stack. Please notice that the push and pop of the ***SP*** register is not valid.
`pop8 %`***register***	`10001`***rrr***	
`push16 %`***register***	`10010`***rrr***	Push on top of the stack the 16-bit content of the specified register, otherwise restore the specified register removing the last 16-bit value from the top of the stack. Please notice that the push and pop of the ***SP*** register is not valid.
`pop16 %`***register***	`10011`***rrr***	
`c8to16u`	`10100000`	Change the ***A*** value to a 8-bit unsigned or signed.
`c8to16s`	`10100001`	
`equal`	`10100010`	Logical operation starting from the values contained inside the registers ***A*** and ***B***, placing the result back to the ***A*** register.
`not`	`10100011`	
`and`	`10100100`	
`nand`	`10100101`	
`or`	`10100110`	
`nor`	`10100111`	
`xor`	`101001000`	
`nxor`	`10101001`	

Macrocode syntax	Binary opcode	Description
`add` `sub`	`10101010` `10101011`	Addition or subtraction: ***A+B*** or ***A−B***. The previous carry or borrow is not taken into consideration, but the result will update the new carry or borrow. The result is saved inside the ***A*** register.

`addc8`
`subb8`
`addc16`
`subb16`
	`10101100`
	`10101101`
	`10101110`
	`10101111`
		Addition or subtraction with previous carry or borrow: ***A+B+carry*** or ***A−B−borrow***. The result is saved inside the ***A*** register. The number eight or sixteen is used to select the input carry or borrow.

`lshl8`
`lshr8`
`ashl8`
`ashr8`
`rotcl8`
`rotcr8`
`rotl8`
`rotr8`
	`10110000`
	`10110001`
	`10110010`
	`10110011`
	`10110100`
	`10110101`
	`10110110`
	`10110111`
		Bit shift and rotation with 8-bit size, over the original ***A*** register value, placing the result on the same ***A*** register.

Macrocode syntax	Binary opcode	Description
`lshl16`	`10111000`	Bit shift and rotation with 16-bit size, over the original ***A*** register value, placing the result on the same ***A*** register.
`lshr16`	`10111001`	
`ashl16`	`10111010`	
`ashr16`	`10111011`	
`rotcl16`	`10111100`	
`rotcr16`	`10111101`	
`rotl16`	`10111110`	
`rotr16`	`10111111`	
`in` ***io***	`11000000`	Read or write a value at the I/O address specified. The I/O address is a 8-bit number. For **`is_ack`** it is checked the confirmation (acknowledge) from the device and, if there is one, the condition is verified and a jump is done to the address specified as the last argument. Currently, only the screen device can be used with the **`ifack_jump`** instruction.
`out` ***io***	`11000001`	
`ifack_jump` ***io*** `,` `#`***ref***	`11000010`	
`clearc`	`11000100`	Clear or set the carry flags inside the ***FL*** register.
`setc`	`11000101`	

Macrocode syntax	Binary opcode	Description

`cmp`
 	
`test`
	`11000110`
	
	`11000111`
		Compare registers ***A*** and ***B***, simulating a subtraction or the AND, just to update the flags inside the ***FL*** register.

`imrl` ***mask***	`11001000`	Load the hardware interrupt mask: the active bits are related to interrupts to be enabled. The argument is a 8-bit number, but only the least significant four bits are used (there are only four hardware interrupts).
`inc %`***register*** `dec %`***register***	`11010`***rrr*** `11011`***rrr***	One unit increment or decrement, to the specified register, which is written at the last tree bits inside the opcode.
`stop`	`11111111`	CPU halt, locking the clock signal.

10.10 Microcode

«

Listing 10.38. Fields of the microcode word, for TKGate.

```
field ctrl[1:0]     = {nop=0, stop=1, load=2};
field pc[10:2]      = {br=1, bw=2, aux=4, low=8, high=16,
                       p1=32, p2=64, m1=128, m2=256};
field sel[15:11]    = {if_carry_8=1, if_not_carry_8=3,
                       if_zero_8=5, if_not_zero_8=7,
                       if_negative_8=9, if_not_negative_8=11,
                       if_overflow_8=13, if_not_overflow_8=15,
                       if_carry_16=1, if_not_carry_16=3,
                       if_zero_16=5, if_not_zero_16=7,
                       if_negative_16=9, if_not_negative_16=11,
                       if_overflow_16=13, if_not_overflow_16=15};
field mdr[24:16]    = {br=1, bw=2, aux=4, low=8, high=16,
                       p1=32, p2=64, m1=128, m2=256};
field i[33:25]      = {br=1, bw=2, aux=4, low=8, high=16,
```

```
                        p1=32, p2=64, m1=128, m2=256};
field j[42:34]     = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field ram[47:43]   = {br=1, bw=2, aux=4, p=0, i=8, j=16, s=24};
field ivt[50:48]   = {br=1, bw=2, inta=0, intb=4};
field ir[59:51]    = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field bus[77:60]   = {bw=0x10000, aux=0x20000};
field irq[80:78]   = {br=1, bw=2, done=4};
field sp[89:81]    = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field bp[98:90]    = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field b[107:99]    = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field fl[116:108]  = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field alu[127:117] = {bw=1, aux=2, sign=4, rank8=0, rank16=8,
                        a=0, and=16, or=32, xor=48,
                        nxor=64, nor=80, nand=96, not=112,
                        add=256, sub=228, addc=320, subb=352,
                        lshl=512, lshr=528, ashl=484, ashr=560,
                        rotcl=576, rotcr=592,
                        rotl=768, rotr=784,
                        clearc=1024, clearz=1040, clearn=1056,
                        clearo=1072, cleari=1088,
                        setc=1152, setz=1168, setn=1184,
                        seto=1200, seti=1216};
field a[136:128]   = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field ioa[145:137] = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field ioc[149:146] = {br=1, bw=2, req=4, isack=8};
```

Listing 10.39. Microcode.

```
begin microcode @ 0
//
// fetch:
//       IR  <-- RAM[pc++]; load;
//
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
//
nop:
    // fetch:
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
op_error:
    // INT 0
    // push FL
    sp=m2;                                      // SP <-- (SP - 2)
    ram=br ram=s fl=bw fl=low  sp=p1;           // RAM[sp++] <- FL[7:0];
    ram=br ram=s fl=bw fl=high sp=m1;           // RAM[sp--] <- FL[15:8];
    // reset interrupt enable flag
    fl=br fl=aux alu=cleari;
    // push PC
    sp=m2;                                      // SP <-- (SP - 2)
    ram=br ram=s pc=bw pc=low  sp=p1;           // RAM[sp++] <- PC[7:0];
    ram=br ram=s pc=bw pc=high sp=m1;           // RAM[sp--] <- PC[15:8];
    // push I
    sp=m2;                                      // SP <-- (SP - 2)
    ram=br ram=s i=bw i=low  sp=p1;             // RAM[sp++] <- I[7:0];
    ram=br ram=s i=bw i=high sp=m1;             // RAM[sp--] <- I[15:8];
    //
    i=br ivt=bw ivt=intb  bus=bw bus=aux bus=0;     // I <- IVT <- 0;
    pc=br pc=low  ram=bw ram=i i=p1;            // PC[7:0]  <-- RAM[i++]
    pc=br pc=high ram=bw ram=i i=m1;            // PC[15:7] <-- RAM[i--]
    // pop I
    i=br i=low  ram=bw ram=s sp=p1;             // I[7:0] <-- RAM[sp++];
    i=br i=high ram=bw ram=s sp=p1;             // I[15:0] <-- RAM[sp++];
    // fetch
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_j:
    j=br i=bw                                   // J <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_a:
    a=br i=bw                                   // A <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
```

```
mv_i_b:
    b=br i=bw                                   // B <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_bp:
    bp=br i=bw                                  // BP <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_sp:
    sp=br i=bw                                  // SP <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_mdr:
    mdr=br i=bw                                 // MDR <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_fl:
    fl=br i=bw                                  // FL <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_i:
    i=br j=bw                                   // I <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_a:
    a=br j=bw                                   // A <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_b:
    b=br j=bw                                   // B <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_bp:
    bp=br j=bw                                  // BP <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_sp:
    sp=br j=bw                                  // SP <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_mdr:
    mdr=br j=bw                                 // MDR <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_fl:
    fl=br j=bw                                  // FL <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_i:
    i=br a=bw                                   // I <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_j:
    j=br a=bw                                   // J <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
```

```
mv_a_b:
    b=br a=bw                                   // B <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_bp:
    bp=br a=bw                                  // BP <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_sp:
    sp=br a=bw                                  // SP <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_mdr:
    mdr=br a=bw                                 // MDR <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_fl:
    fl=br a=bw                                  // FL <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_i:
    i=br b=bw                                   // I <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_j:
    j=br b=bw                                   // J <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_a:
    a=br b=bw                                   // A <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_bp:
    bp=br b=bw                                  // BP <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_sp:
    sp=br b=bw                                  // SP <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_mdr:
    mdr=br b=bw                                 // MDR <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_fl:
    fl=br b=bw                                  // FL <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_i:
    i=br bp=bw                                  // I <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_j:
    j=br bp=bw                                  // J <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
```

```
mv_bp_a:
    a=br bp=bw                                  // A <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_b:
    b=br bp=bw                                  // B <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_sp:
    sp=br bp=bw                                 // SP <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_mdr:
    mdr=br bp=bw                                // MDR <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_fl:
    fl=br bp=bw                                 // FL <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_i:
    i=br sp=bw                                  // I <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_j:
    j=br sp=bw                                  // J <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_a:
    a=br sp=bw                                  // A <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_bp:
    bp=br sp=bw                                 // BP <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_b:
    b=br sp=bw                                  // B <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_mdr:
    mdr=br sp=bw                                // MDR <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_fl:
    fl=br sp=bw                                 // FL <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_i:
    i=br mdr=bw                                 // I <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_j:
    j=br mdr=bw                                 // J <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
```

```
mv_mdr_bp:
    bp=br mdr=bw                                  // BP <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_sp:
    sp=br mdr=bw                                  // SP <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_b:
    b=br mdr=bw                                   // B <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_a:
    a=br mdr=bw                                   // A <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_fl:
    fl=br mdr=bw                                  // FL <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_i:
    i=br fl=bw                                    // I <- FL, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_j:
    j=br fl=bw                                    // J <- FL, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_a:
    a=br fl=bw                                    // A <- FL, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_bp:
    bp=br fl=bw                                   // BP <- FL, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_sp:
    sp=br fl=bw                                   // SP <- FL, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_mdr:
    mdr=br fl=bw                                  // MDR <- FL, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_b:
    fl=br b=bw                                    // B <- FL, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
load8_i:
    mdr=br ram=bw ram=i;                          // MDR <- RAM[i];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
load8_j:
    mdr=br ram=bw ram=j;                          // MDR <- RAM[j];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
```

```
load8:
    i=br i=low  ram=bw ram=p pc=p1;          // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;          // I[15:8] <- RAM[pc++];
    mdr=br ram=bw ram=i;                     // MDR <- RAM[i];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
load16_i:
    mdr=br mdr=low  ram=bw ram=i i=p1;       // MDR[7:0] <- RAM[i++];
    mdr=br mdr=high ram=bw ram=i i=m1;       // MDR[15:8] <- RAM[i--];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
load16_j:
    mdr=br mdr=low  ram=bw ram=j j=p1;       // MDR[7:0] <- RAM[j++];
    mdr=br mdr=high ram=bw ram=j j=m1;       // MDR[15:8] <- RAM[j--];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
load16:
    i=br i=low  ram=bw ram=p pc=p1;          // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;          // I[15:8] <- RAM[pc++];
    mdr=br mdr=low  ram=bw ram=i i=p1;       // MDR[7:0] <- RAM[i++];
    mdr=br mdr=high ram=bw ram=i i=m1;       // MDR[15:8] <- RAM[i--];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store8_i:
    ram=br ram=i mdr=bw;                     // RAM[i] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store8_j:
    ram=br ram=j mdr=bw;                     // RAM[j] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store8:
    i=br i=low  ram=bw ram=p pc=p1;          // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;          // I[15:8] <- RAM[pc++];
    ram=br ram=i mdr=bw;                     // RAM[i] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store16_i:
    ram=br ram=i mdr=bw mdr=low  i=p1;       // RAM[i++] <- MDR[7:0];
    ram=br ram=i mdr=bw mdr=high i=m1;       // RAM[i--] <- MDR[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store16_j:
    ram=br ram=j mdr=bw mdr=low  j=p1;       // RAM[j++] <- MDR[7:0];
    ram=br ram=j mdr=bw mdr=high j=m1;       // RAM[j--] <- MDR[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store16:
    i=br i=low  ram=bw ram=p pc=p1;          // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;          // I[15:8] <- RAM[pc++];
    ram=br ram=i mdr=bw mdr=low  i=p1;       // RAM[i++] <- MDR[7:0];
```

```
    ram=br ram=i mdr=bw mdr=high i=m1;      // RAM[i--] <- MDR[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cp8_ij:
    mdr=br ram=bw ram=i i=p1;               // MDR[7:0] <- RAM[i++];
    ram=br ram=j mdr=bw j=p1;               // RAM[j++] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cp8_ji:
    mdr=br ram=bw ram=j j=p1;               // MDR[7:0] <- RAM[j++];
    ram=br ram=i mdr=bw i=p1;               // RAM[i++] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cp16_ij:
    mdr=br mdr=low  ram=bw ram=i i=p1;      // MDR[7:0] <- RAM[i++];
    mdr=br mdr=high ram=bw ram=i i=p1;      // MDR[15:8] <- RAM[i++];
    ram=br ram=j mdr=bw mdr=low  j=p1;      // RAM[j++] <- MDR[7:0];
    ram=br ram=j mdr=bw mdr=high j=p1;      // RAM[j++] <- MDR[15:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cp16_ji:
    mdr=br mdr=low  ram=bw ram=j j=p1;      // MDR[7:0] <- RAM[j++];
    mdr=br mdr=high ram=bw ram=j j=p1;      // MDR[15:8] <- RAM[j++];
    ram=br ram=i mdr=bw mdr=low  i=p1;      // RAM[i++] <- MDR[7:0];
    ram=br ram=i mdr=bw mdr=high i=p1;      // RAM[i++] <- MDR[15:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump:
    i=br pc=bw;                             // I <- PC
    pc=br pc=low  ram=bw ram=i i=p1;        // PC[7:0]  <-- RAM[i++]
    pc=br pc=high ram=bw ram=i i=m1;        // PC[15:7] <-- RAM[i--]
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
jump8c:
    mdr=br mdr=low  ram=bw ram=p pc=p1;     // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;     // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_carry_8;                   // PC = (carry8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8nc:
    mdr=br mdr=low  ram=bw ram=p pc=p1;     // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;     // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_carry_8;               // PC = (not_carry8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8z:
    mdr=br mdr=low  ram=bw ram=p pc=p1;     // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;     // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_zero_8;                    // PC = (zero8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
```

```
jump8nz:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_zero_8;                 // PC = (not_zero8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8o:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_overflow_8;                 // PC = (overflow8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8no:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_overflow_8;             // PC = (not_overflow8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8n:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_negative_8;                 // PC = (negative8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8nn:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_negative_8;             // PC = (not_negative8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16c:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_carry_16;                   // PC = (carry16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16nc:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_carry_16;               // PC = (not_carry16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16z:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_zero_16;                    // PC = (zero16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16nz:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
```

```
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_zero_16;                // PC = (not_zero16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16o:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_overflow_16;                // PC = (overflow16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16no:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_overflow_16;            // PC = (not_overflow16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16n:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_negative_16;                // PC = (negative16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16nn:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_negative_16;            // PC = (not_negative16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
call:
    i=br i=low  ram=bw ram=p pc=p1 sp=m1;    // I[7:0] <-- RAM[pc++], SP--
    i=br i=high ram=bw ram=p pc=p1 sp=m1;    // I[15:7] <-- RAM[pc++], SP--
    ram=br ram=s pc=bw pc=low  sp=p1;        // RAM[sp++] <- PC[7:0], SP++
    ram=br ram=s pc=bw pc=high sp=m1;        // RAM[sp--] <- PC[15:8], SP--
    pc=br i=bw;                              // PC <- I;
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
call_i:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s pc=bw pc=low  sp=p1;        // RAM[sp++] <- PC[7:0], SP++
    ram=br ram=s pc=bw pc=high sp=m1;        // RAM[sp--] <- PC[15:8], SP--
    pc=br i=bw;                              // PC <- I;
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
call_j:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s pc=bw pc=low  sp=p1;        // RAM[sp++] <- PC[7:0], SP++
    ram=br ram=s pc=bw pc=high sp=m1;        // RAM[sp--] <- PC[15:8], SP--
    pc=br j=bw;                              // PC <- J;
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
```

```
return:
    pc=br pc=low  ram=bw ram=s sp=p1;        // PC[7:0] <- RAM[sp++];
    pc=br pc=high ram=bw ram=s sp=p1;        // PC[15:8] <- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_i:
    sp=m1;                                   // SP--;
    ram=br ram=s i=bw i=low;                 // RAM[sp] <- I[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_j:
    sp=m1;                                   // SP--;
    ram=br ram=s j=bw j=low;                 // RAM[sp] <- J[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_a:
    sp=m1;                                   // SP--;
    ram=br ram=s a=bw a=low;                 // RAM[sp] <- A[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_b:
    sp=m1;                                   // SP--;
    ram=br ram=s b=bw b=low;                 // RAM[sp] <- B[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_bp:
    sp=m1;                                   // SP--;
    ram=br ram=s bp=bw bp=low;               // RAM[sp] <- BP[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_mdr:
    sp=m1;                                   // SP--;
    ram=br ram=s mdr=bw mdr=low;             // RAM[sp] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_fl:
    sp=m1;                                   // SP--;
    ram=br ram=s fl=bw fl=low;               // RAM[sp] <- FL[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_i:
    i=br i=low  ram=bw ram=s sp=p1           // I[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_j:
    j=br j=low  ram=bw ram=s sp=p1;          // J[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_a:
    a=br a=low  ram=bw ram=s sp=p1;          // A[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_b:
```

```
    b=br b=low  ram=bw ram=s sp=p1;          // B[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_bp:
    bp=br bp=low  ram=bw ram=s sp=p1;        // BP[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_mdr:
    mdr=br mdr=low  ram=bw ram=s sp=p1;      // MDR[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_fl:
    fl=br fl=low  ram=bw ram=s sp=p1;        // FL[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_i:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s i=bw i=low  sp=p1;          // RAM[sp++] <- I[7:0];
    ram=br ram=s i=bw i=high sp=m1;          // RAM[sp--] <- I[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_j:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s j=bw j=low  sp=p1;          // RAM[sp++] <- J[7:0];
    ram=br ram=s j=bw j=high sp=m1;          // RAM[sp--] <- J[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_a:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s a=bw a=low  sp=p1;          // RAM[sp++] <- A[7:0];
    ram=br ram=s a=bw a=high sp=m1;          // RAM[sp--] <- A[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_b:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s b=bw b=low  sp=p1;          // RAM[sp++] <- B[7:0];
    ram=br ram=s b=bw b=high sp=m1;          // RAM[sp--] <- B[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_bp:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s bp=bw bp=low  sp=p1;        // RAM[sp++] <- BP[7:0];
    ram=br ram=s bp=bw bp=high sp=m1;        // RAM[sp--] <- BP[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_mdr:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s mdr=bw mdr=low  sp=p1;      // RAM[sp++] <- MDR[7:0];
    ram=br ram=s mdr=bw mdr=high sp=m1;      // RAM[sp--] <- MDR[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_fl:
```

```
    sp=m2;                                      // SP <-- (SP - 2)
    ram=br ram=s fl=bw fl=low  sp=p1;           // RAM[sp++] <- FL[7:0];
    ram=br ram=s fl=bw fl=high sp=m1;           // RAM[sp--] <- FL[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_i:
    i=br i=low  ram=bw ram=s sp=p1;             // I[7:0] <-- RAM[sp++];
    i=br i=high ram=bw ram=s sp=p1;             // I[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_j:
    j=br j=low  ram=bw ram=s sp=p1;             // J[7:0] <-- RAM[sp++];
    j=br j=high ram=bw ram=s sp=p1;             // J[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_a:
    a=br a=low  ram=bw ram=s sp=p1;             // A[7:0] <-- RAM[sp++];
    a=br a=high ram=bw ram=s sp=p1;             // A[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_b:
    b=br b=low  ram=bw ram=s sp=p1;             // B[7:0] <-- RAM[sp++];
    b=br b=high ram=bw ram=s sp=p1;             // B[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_bp:
    bp=br bp=low  ram=bw ram=s sp=p1;           // BP[7:0] <-- RAM[sp++];
    bp=br bp=high ram=bw ram=s sp=p1;           // BP[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_mdr:
    mdr=br mdr=low  ram=bw ram=s sp=p1;         // MDR[7:0] <-- RAM[sp++];
    mdr=br mdr=high ram=bw ram=s sp=p1;         // MDR[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_fl:
    fl=br fl=low  ram=bw ram=s sp=p1;           // FL[7:0] <-- RAM[sp++];
    fl=br fl=high ram=bw ram=s sp=p1;           // FL[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
c8to16u:
    a=br alu=bw alu=a alu=rank8 fl=br fl=aux        // A[15:0] <- A[7:0],
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch;
c8to16s:
    a=br alu=bw alu=a alu=rank8 alu=sign fl=br fl=aux   // A[15:0] <- A[7:0],
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch;
equal:
    a=br alu=bw alu=a alu=rank16 fl=br fl=aux       // A <- A, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
not:
```

```
    a=br alu=bw alu=not alu=rank16 fl=br fl=aux     // A <- NOT A, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
and:
    a=br alu=bw alu=and alu=rank16 fl=br fl=aux     // A <- A AND B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
nand:
    a=br alu=bw alu=nand alu=rank16 fl=br fl=aux    // A <- A NAND B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
or:
    a=br alu=bw alu=or alu=rank16 fl=br fl=aux      // A <- A OR B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
nor:
    a=br alu=bw alu=nor alu=rank16 fl=br fl=aux     // A <- A NOR B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
xor:
    a=br alu=bw alu=xor alu=rank16 fl=br fl=aux     // A <- A XOR B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
nxor:
    a=br alu=bw alu=nxor alu=rank16 fl=br fl=aux    // A <- A NXOR B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
add:
    a=br alu=bw alu=add alu=rank16 fl=br fl=aux     // A <- A+B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
sub:
    a=br alu=bw alu=sub alu=rank16 fl=br fl=aux     // A <- A-B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
addc8:
    a=br alu=bw alu=addc alu=rank8 fl=br fl=aux     // A <- A+B+carry, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
subb8:
    a=br alu=bw alu=subb alu=rank8 fl=br fl=aux     // A <- A-B-borrow, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
addc16:
    a=br alu=bw alu=addc alu=rank16 fl=br fl=aux    // A <- A+B+carry, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
subb16:
    a=br alu=bw alu=subb alu=rank16 fl=br fl=aux    // A <- A-B-borrow, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
lshl8:
    a=br alu=bw alu=lshl alu=rank8 fl=br fl=aux     // A <- A <<, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
lshr8:
```

```
    a=br alu=bw alu=lshr alu=rank8 fl=br fl=aux    // A <- A >>, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
ashl8:
    a=br alu=bw alu=ashl alu=rank8 alu=sign fl=br fl=aux    // A <- A*2,
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch:
ashr8:
    a=br alu=bw alu=ashr alu=rank8 alu=sign fl=br fl=aux   // A <- A/2, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotcl8:
    a=br alu=bw alu=rotcl alu=rank8 fl=br fl=aux   // A <- rotcl(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotcr8:
    a=br alu=bw alu=rotcr alu=rank8 fl=br fl=aux   // A <- rotcr(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotl8:
    a=br alu=bw alu=rotl alu=rank8 fl=br fl=aux    // A <- rotl(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotr8:
    a=br alu=bw alu=rotr alu=rank8 fl=br fl=aux    // A <- rotr(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
lshl16:
    a=br alu=bw alu=lshl alu=rank16 fl=br fl=aux   // A <- A <<, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
lshr16:
    a=br alu=bw alu=lshr alu=rank16 fl=br fl=aux   // A <- A >>, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
ashl16:
    a=br alu=bw alu=ashl alu=rank16 alu=sign fl=br fl=aux    // A <- A*2,
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch;
ashr16:
    a=br alu=bw alu=ashr alu=rank16 alu=sign fl=br fl=aux    // A <- A/2,
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch;
rotcl16:
    a=br alu=bw alu=rotcl alu=rank16 fl=br fl=aux  // A <- rotcl(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotcr16:
    a=br alu=bw alu=rotcr alu=rank16 fl=br fl=aux  // A <- rotcr(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotl16:
    a=br alu=bw alu=rotl alu=rank16 fl=br fl=aux   // A <- rotl(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotr16:
```

```
    a=br alu=bw alu=rotr alu=rank16 fl=br fl=aux   // A <- rotr(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
in:
    ioa=br ram=bw ram=p pc=p1;                    // IOA <- RAM[pc++];
    ioc=req;                                      // I/O request;
    ctrl=nop;                                     // does not do anything
    a=br ioc=bw                                   // A <- I/O, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
out:
    ioa=br ram=bw ram=p pc=p1;                    // IOA <- RAM[pc++];
    ioc=br a=bw;                                  // I/O <- A
    ioc=req                                       // I/O request, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
ifack_jump:
    ioa=br ram=bw ram=p pc=p1;                    // IOA <- RAM[pc++];
    mdr=br mdr=low  ram=bw ram=p pc=p1;           // MDR[7:0] <- RAM[pc++];
    mdr=br mdr=high ram=bw ram=p pc=p1;           // MDR[15:8] <- RAM[pc++];
    a=br ioc=bw ioc=isack;                        // A <- I/O is ack;
    a=br alu=a alu=rank8 alu=sign fl=br fl=aux; // A[15:0] <- A[7:0];
    pc=br sel=if_not_zero_8;                      // PC = (not_zero8?MDR:PC);
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
int:
    // push FL
    sp=m2;                                        // SP <-- (SP - 2)
    ram=br ram=s fl=bw fl=low  sp=p1;             // RAM[sp++] <- FL[7:0];
    ram=br ram=s fl=bw fl=high sp=m1;             // RAM[sp--] <- FL[15:8];
    // reset interrupt enable flag, PC++
    // (PC is incremented to jump the argument, before it is saved
    // inside the stack).
    fl=br fl=aux alu=cleari pc=p1;
    // push PC
    sp=m2;                                        // SP <-- (SP - 2)
    ram=br ram=s pc=bw pc=low  sp=p1;             // RAM[sp++] <- PC[7:0];
    ram=br ram=s pc=bw pc=high sp=m1;             // RAM[sp--] <- PC[15:8];
    // push I
    sp=m2;                                        // SP <-- (SP - 2)
    ram=br ram=s i=bw i=low  sp=p1;               // RAM[sp++] <- I[7:0];
    ram=br ram=s i=bw i=high sp=m1;               // RAM[sp--] <- I[15:8];
    // Set the PC register back so that the argument
    // can be read.
    pc=m1;
    //
```

```
    i=br ivt=bw ivt=intb ram=bw ram=aux ram=p pc=p1; // I <- IVT <- RAM[pc++];
    pc=br pc=low  ram=bw ram=i i=p1;         // PC[7:0]  <-- RAM[i++]
    pc=br pc=high ram=bw ram=i i=m1;         // PC[15:7] <-- RAM[i--]
    // pop I
    i=br i=low  ram=bw ram=s sp=p1;          // I[7:0] <-- RAM[sp++];
    i=br i=high ram=bw ram=s sp=p1;          // I[15:0] <-- RAM[sp++];
    // fetch
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
iret:
    // pop PC
    pc=br pc=low  ram=bw ram=s sp=p1;        // PC[7:0] <- RAM[sp++];
    pc=br pc=high ram=bw ram=s sp=p1;        // PC[15:8] <- RAM[sp++];
    // pop FL
    fl=br fl=low  ram=bw ram=s sp=p1;        // FL[7:0] <-- RAM[sp++];
    fl=br fl=high ram=bw ram=s sp=p1;        // FL[15:0] <-- RAM[sp++];
    // fetch
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
irq:
    // push FL
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s fl=bw fl=low  sp=p1;        // RAM[sp++] <- FL[7:0];
    ram=br ram=s fl=bw fl=high sp=m1;        // RAM[sp--] <- FL[15:8];
    // reset interrupt enable flag
    fl=br fl=aux alu=cleari;
    // Restore the PC correct value: PC is currently located after the
    // opcode of an instruction that is not currently executed,
    // because there is a hardware interrupt to serve; that is why the
    // correct value to save inside the stack is PC-1.
    pc=m1;                                   // PC--;
    // push PC
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s pc=bw pc=low  sp=p1;        // RAM[sp++] <- PC[7:0];
    ram=br ram=s pc=bw pc=high sp=m1;        // RAM[sp--] <- PC[15:8];
    // push I
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s i=bw i=low  sp=p1;          // RAM[sp++] <- I[7:0];
    ram=br ram=s i=bw i=high sp=m1;          // RAM[sp--] <- I[15:8];
    //
    i=br ivt=bw ivt=inta;                    // I <- IVT <- IRQ;
    pc=br pc=low  ram=bw ram=i i=p1;         // PC[7:0]  <-- RAM[i++]
    pc=br pc=high ram=bw ram=i i=m1;         // PC[15:7] <-- RAM[i--]
    // pop I
```

```
    i=br i=low  ram=bw ram=s sp=p1;          // I[7:0] <-- RAM[sp++];
    i=br i=high ram=bw ram=s sp=p1;          // I[15:0] <-- RAM[sp++];
    //
    irq=done;
    // fetch
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
ivtl:
    i=br i=low  ram=bw ram=p pc=p1;          // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;          // I[15:8] <- RAM[pc++];
    ivt=br i=bw;                             // IVT <- MDR;
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
imrl:
    irq=br ram=bw ram=p pc=p1;               // IRQ <- RAM[pc++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cleari:
    fl=br fl=aux alu=cleari irq=done
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
seti:
    fl=br fl=aux alu=seti irq=done
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
clearc:
    fl=br fl=aux alu=clearc
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
setc:
    fl=br fl=aux alu=setc
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
cmp:
    fl=br fl=aux alu=sub                     // FL(A - B);
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
test:
    fl=br fl=aux alu=and                     // FL(A AND B);
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_i:
    i=p1                                     // I++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_j:
    j=p1                                     // J++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_a:
    a=p1                                     // A++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_b:
```

```
    b=p1                                        // B++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_bp:
    bp=p1                                       // BP++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_sp:
    sp=p1                                       // SP++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_mdr:
    mdr=p1                                      // MDR++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_fl:
    fl=p1                                       // FL++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_i:
    i=m1                                        // I--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_j:
    j=m1                                        // J--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_a:
    a=m1                                        // A--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_b:
    b=m1                                        // B--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_bp:
    bp=m1                                       // BP--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_sp:
    sp=m1                                       // SP--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_mdr:
    mdr=m1                                      // MDR--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_fl:
    fl=m1                                       // FL--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
//
stop:
    ctrl=stop;                                      // stop clock
    // if resumed, fetch:
```

```
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
//
end
```

10.11 Interrupts management

There are three types of interrupts: those generated internally by the CPU, those generated by the hardware (IRQ) and the software interrupts. Internal interrupts of the CPU can range from INT0 to INT3, but currently there is only INT0 which is related to the identification of a wrong opcode. The hardware interrupts can range from INT4 to INT7 and match respectively the range IRQ0 to IRQ3. Software interrupts can range from INT8 to INT15. The IVT table (*interrupt vector table*) must be prepared through the macrocode, which must initialize the ***IVT*** register with the address of the table location, as in the following example:

```
begin macrocode @ 0
        jump #start
        nop
interrupt_vector_table:
        .short  0x0025  // CPU      # op_code_error
        .short  0x0024  // CPU      # default_interrupt_routine
        .short  0x0024  // CPU      # default_interrupt_routine
        .short  0x0024  // CPU      # default_interrupt_routine
        .short  0x0024  // IRQ      # default_interrupt_routine
        .short  0x0024  // IRQ      # default_interrupt_routine
        .short  0x0024  // IRQ      # default_interrupt_routine
        .short  0x0024  // IRQ      # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
default_interrupt_routine:
```

```
        iret
op_code_error:
        stop
start:
        load16 #sp_base
        mv %MDR, %SP
        ivtl #interrupt_vector_table
        imrl 0x0F // tutti
        seti
        …
        …
sp_base:
        .short 0x0080
end
```

The table might also be shorter, if there is no need for the software interrupts.

The occurrence of an interrupt (CPU, hardware or software) involves the following: the ***FL*** register is saved on top of the stack, the interrupt enable flag is disabled, the ***PC*** register is saved on top of the stack. The **`iret`** instruction restores the ***PC*** and then the ***FL*** registers. But while the conclusion of an interrupt takes place always in the same way, through the description of the opcode **`iret`**, the start is different in the three cases. If it comes to interruption due to a wrong opcode, the microcode routine at the label ***op_error*** is performed; if it is a hardware interrupt, the microcode routine starting at the ***irq*** label is run, as specified by the dip-switch inside the control unit; if it is a software interrupt, the microcode routine starting at the ***int*** label is executed. The three situations are different:

- The interrupt due to an incorrect opcode depends on an error from the code read inside the RAM memory, but it is not known the extent of this damage. Not being able to guess, the best choice for the routine associated to the interruption should coincide with the CPU halt, otherwise, it might be the case to try to ignore the wrong opcode and just try the next memory cell, without being able to know if there is possibly an argument to the wrong opcode just read.
- The hardware interrupt (IRQ) occurs asynchronously with respect to the CPU activity and it is served only when the CPU itself would be ready to execute a new

opcode. In this condition, the ***PC*** register already points to the memory location next to the opcode that would have to be executed, so, before saving the ***PC*** register inside the stack, it is necessary to move back the ***PC*** one position: this way it points to the position of the opcode that must be executed after the interrupt is served. The microcode related to a hardware interrupt has also the task, once read the address matching the interrupt from the IVT table, to cancel the pending request from the **IRQ** module. This is done by sending a signal through the control bus, which in the **IRQ** module is implemented as ***irq_done***. It is then a task of the logic inside the **IRQ** module to know what is actually the IRQ signal to reset. At the same time, the **IRQ** module might require the management of another interrupt, but this management would be temporarily suspended, because the interrupt enable flag is disabled (it is disabled immediately after the ***FL*** register is saved on the stack).

- The software interrupt is easier to manage, because it happens in a predictable way, without really interrupting the control unit activity.

When an interruption occurs also exists the need to jump properly to the routine provided in the IVT table. The ***IVT*** register has two separate inputs to receive the interrupt number to convert to memory address: one is connected to auxiliary bus, which is also connected to the **BUS** module and the **RAM** module; the other is connected to the **IRQ** module. When a CPU internal interrupt is generated, the **IVT** module is controlled directly by the control unit, through the **BUS** module; when a hardware interrupt is received, the **IVT** module is driven by the **IRQ** module; when it is a software interrupt, the **IVT** module receives the interrupt number from the RAM, as it is supplied as an argument to the opcode. It should also be noted that with the software interrupt, any interrupt number can be specified.

10.12 Module "RTC": real time clock

«

The **RTC** module (real time clock) produces one pulse per second through the hardware interrupt IRQ0. If the IRQ0 is enabled inside the **IRQ** module, an INT4 is requested at every second. INT4 is translated into the fifth position inside the IVT table. The **RTC** module is described only by Verilog code:

Listing 10.41. Module **RTC**.

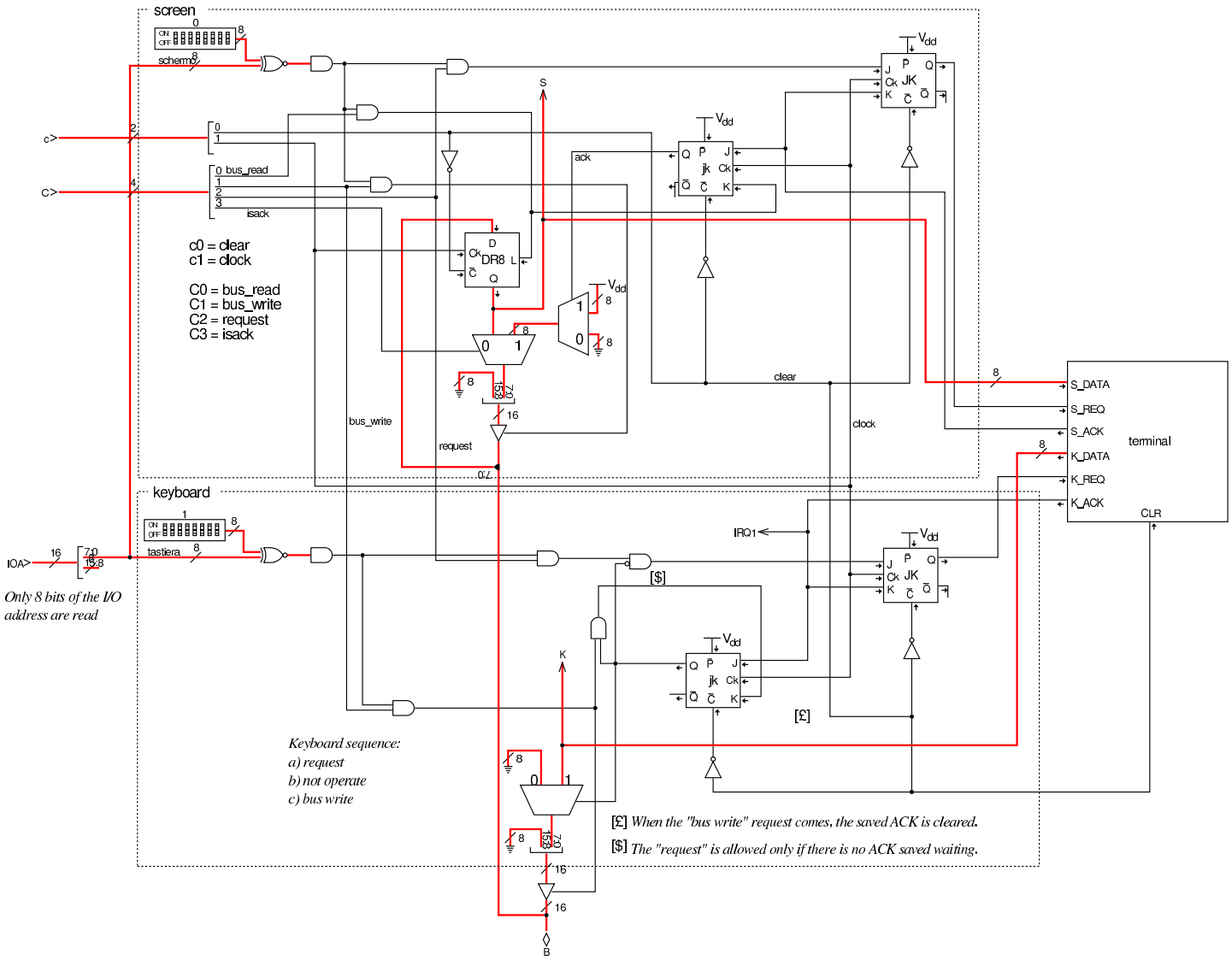
```
module RTC (T);
  output T;
  reg p;
  always
    begin
      p = 0;
      $tkg$wait (500);
      p = 1;
      $tkg$wait (500);
    end
  assign T = p;
endmodule
```

10.13 Module "TTY"

«

The module TTY, for the management of the video-keyboard terminal, is almost the same as the previous version of the project: it adds an output connected to the signal of confirmation (acknowledge) of the keyboard, for controlling the signal IRQ1. In this way, when the user press a key on the keyboard, an interrupt IRQ1 is produced: the routine associated to the hardware interrupt can be used to execute the code necessary to handle what was typed.

Figure 10.42. Module **TTY**.



Listing 10.43. Verilog code describing the module **terminal**.

```
module terminal(K_DATA, K_REQ, K_ACK, S_DATA, S_REQ, S_ACK, CLR);

output K_ACK;
output S_ACK;
output [7:0] K_DATA;
input [7:0] S_DATA;
input K_REQ;
input S_REQ;
input CLR;
reg k_ready;
reg [7:0] key;
```

```
reg s_ready;


  initial
    begin
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  always
    begin
      @(posedge CLR)
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  initial $tkg$post("TERMINAL","%m");

  always
    begin
      @ (posedge K_REQ);
      # 5;
      key = $tkg$recv("%m.KD");
      # 5;
      k_ready = 1'b1;
      # 5;
      @ (negedge K_REQ);
      # 5;
      k_ready = 1'b0;
    end

  always
    begin
      @(posedge S_REQ);
      # 5;
```

```
        $tkg$send("%m.SD",S_DATA);
        # 5;
        s_ready = 1'b1;
        # 5;
        @(negedge S_REQ);
        # 5;
        s_ready = 1'b0;
      end

  assign S_ACK = s_ready;
  assign K_DATA = key;
  assign K_ACK = k_ready;

endmodule
```

Listing 10.44. File '`share/tkgate/vpd/terminal.tcl`' for TCL interface.

```
image create bitmap txtcurs -file "$bd/txtcurs.b"

VPD::register TERMINAL
VPD::allow TERMINAL::post
VPD::allow TERMINAL::data

namespace eval TERMINAL {
    # Public variables declarations: the variables $terminal_...
    # are arrays of which only the element $n is used;
    # that element identifies uniquely the working interface instance.
    variable terminal_w
    variable terminal_pos
    #
    variable KD
    # Function requested by TKGate to create the interface.
    proc post {n} {
        variable terminal_w
        variable terminal_pos
        # Create the window and save the object element in a $terminal_w array element.
        set terminal_w($n) [VPD::createWindow "TERMINAL $n" -shutdowncommand "TERMINAL::unpost $n"]
        # For convenience, copy the object reference inside the local
        # variable $w; then, the variable $w will be used as a reference to the object.
        set w $terminal_w($n)
        text $w.txt -state disabled
        pack $w.txt
        # Put the cursor at the end of the displayed text.
        $w.txt image create end -image txtcurs
        # Bind the keyboard input, related to the object represented by
        # $terminal_w($n), to the function sendChar.
```

```
        bind $w <KeyPress> "TERMINAL::sendChar $n \"%A\""
        # Open a reading channel, named «SD» (screen data),
        # and associate it to the function «data»; moreover, open a
        # writing channel, named «KD» (keyboard data).
        if {[info exists ::tkgate_isInitialized]} {
            VPD::outsignal $n.KD TERMINAL::KD($n)
            VPD::insignal $n.SD -command "TERMINAL::data $n" -format %d
        }
        # Reset the character count, used to count the characters displayed
        # on screen.
        set terminal_pos($n) 0
    }
    # Function that receives the typing and put it into the
    # channel «KD», related to the current interface instance.
    proc sendChar {n key} {
        variable KD
        if { [string length $key ] == 1 } {
            binary scan $key c c
            set TERMINAL::KD($n) $c
        }
    }
    # Function that TKGate requires to destroy the interface.
    proc unpost {n} {
        variable terminal_w
        variable terminal_pos
        destroy $terminal_w($n)
        destroy $terminal_pos($n)
        unset terminal_w($n)
        unset terminal_pos($n)
    }
    # Function used to get the data to display on screen.
    proc data {n c} {
        variable terminal_w
        variable terminal_pos
        # For convenience, copy the object reference representing the
        # interface, inside the variable $w.
        set w $terminal_w($n)
        catch {
            # The variable $c contains the character to display.
            if { $c == 7 } {
                # BEL
                bell
                return
            } elseif { $c == 127 || $c == 8 } {
                # DEL | BS
                if { $terminal_pos($n) > 0 } {
                    # Delete the last displayed character, but only if the
                    # characters counter is greater than zero, otherwise
                    # the cursor would disappear and the next characters
                    # would be located in an unvisible screen area.
                    $w.txt configure -state normal
                    $w.txt delete "end - 3 chars"
```

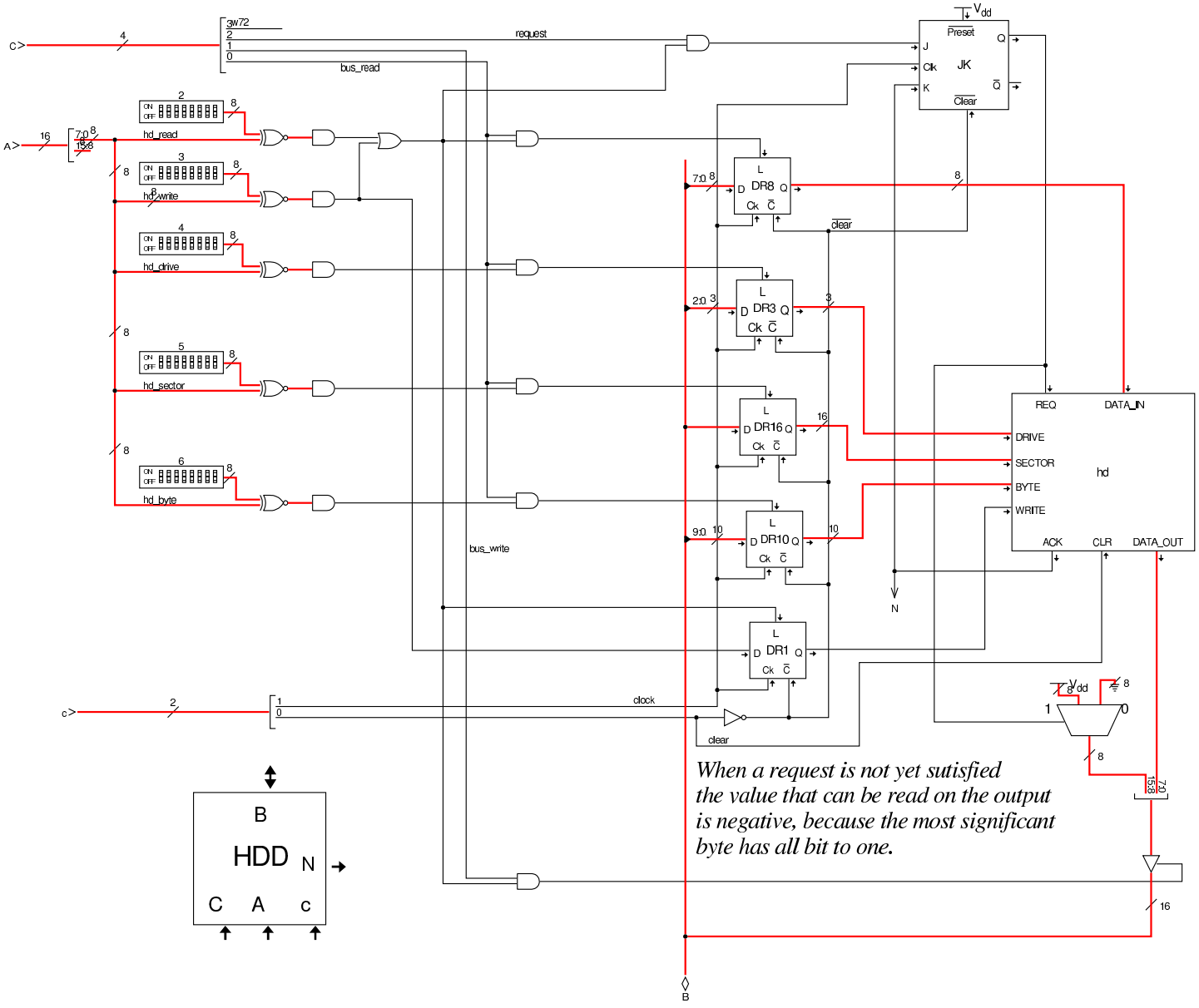
```
                    $w.txt see end
                    $w.txt configure -state disabled
                    set terminal_pos($n) [expr {$terminal_pos($n) - 1}]
                }
                return
            } elseif { $c == 13 } {
                # Convert CR to LF.
                set c 10
            }
            # Convert the character number into a visible symbol.
            set x [format %c $c]
            # Display the symbol.
            $w.txt configure -state normal
            $w.txt insert "end - 2 chars" $x
            $w.txt see end
            $w.txt configure -state disabled
            # Update the displayed characters counter.
            set terminal_pos($n) [expr {$terminal_pos($n) + 1}]
        }
    }
}
```

10.14 Module "HDD"

«

The **HDD** module is new compared to the previous version: it is an interface that simulates a set of eight units of mass storage, each subdivided into sectors of 512 bytes. The device is accessed with different I/O addresses depending on the type of operation that is to be made.

Figure 10.45. Module **HDD**.



Listing 10.46. Verilog code describing the module **hd**.

```
module hd(DRIVE, SECTOR, BYTE, WRITE, DATA_IN, DATA_OUT, REQ, ACK, CLR);
input [2:0] DRIVE;
input WRITE, REQ, CLR;
input [15:0] SECTOR;
input [9:0] BYTE;
input [7:0] DATA_IN;
output [7:0] DATA_OUT;
output ACK;
//
integer _data_out;
```

```
integer _ack;
//
reg [7:0] buffer[0:1023];
reg [8*24-1:0] filename = "hd0_sector_000000000.mem";
//
integer i;
integer sector_8;
integer sector_7;
integer sector_6;
integer sector_5;
integer sector_4;
integer sector_3;
integer sector_2;
integer sector_1;
integer sector_0;
integer x;
//
  initial
    begin
      for (i=0; i<1024; i=i+1)
        begin
          //
          // Initial buffer reset with 00.
          //
          buffer[i] = 8'h00;
        end
      _ack = 0;
      _data_out = 0;
      x = 0;
    end
  //
  always
    begin
      @(posedge CLR)
      _ack = 0;
      _data_out = 0;
      x = 0;
    end
  //
  //
```

```
  //
  always
    begin
      //
      // Start after a positive edge from REQ!.
      //
      @(posedge REQ);
      # 10;
      //
      // Define the sector file name.
      //
      x = SECTOR;
      sector_0 = x%10;
      x = x/10;
      sector_1 = x%10;
      x = x/10;
      sector_2 = x%10;
      x = x/10;
      sector_3 = x%10;
      x = x/10;
      sector_4 = x%10;
      x = x/10;
      sector_5 = x%10;
      x = x/10;
      sector_6 = x%10;
      x = x/10;
      sector_7 = x%10;
      x = x/10;
      sector_8 = x%10;
      //
      // The string starts from right to left!
      //
      filename[12*8+7:12*8] = sector_8 + 8'd48;
      filename[11*8+7:11*8] = sector_7 + 8'd48;
      filename[10*8+7:10*8] = sector_6 + 8'd48;
      filename[9*8+7:9*8] = sector_5 + 8'd48;
      filename[8*8+7:8*8] = sector_4 + 8'd48;
      filename[7*8+7:7*8] = sector_3 + 8'd48;
      filename[6*8+7:6*8] = sector_2 + 8'd48;
      filename[5*8+7:5*8] = sector_1 + 8'd48;
```

```
        filename[4*8+7:4*8] = sector_0 + 8'd48;
        //
        filename[21*8+7:21*8] = DRIVE + 8'd48;
        //
        if (WRITE)
          begin
            //
            // Put data inside the buffer.
            //
            buffer[BYTE] = DATA_IN;
            //
            // Save the buffer to disk.
            // Please remember that $writememh() must be enabled inside
            // TKGate configuration!
            //
            $writememh(filename, buffer);
            //
            // Return the same data read.
            //
            _data_out = buffer[BYTE];
          end
        else
          begin
            //
            // Get data from disk to the buffer.
            //
            $readmemh(filename, buffer);
            //
            // Return the data required.
            //
            _data_out = buffer[BYTE];
          end
        //
        // Acknowledge.
        //
        _ack = 1;
        //
        // Wait the end of request (the negative edge)
        // before restarting the loop.
        //
```

```
        @(negedge REQ);
        # 10;
        //
        // Now become ready again.
        //
        _ack = 0;
      end
      //
    assign DATA_OUT = _data_out;
    assign ACK = _ack;
    //
endmodule
```

Since this is a new module, it is important to describe the behavior of the **hd** module, which has just been shown as a Verilog source: the inputs ***DRIVE***, ***SECTOR*** and ***BYTE*** are used to uniquely identify a byte, belonging to a certain sector of a certain storage device. In practice, each virtual storage unit is divided into sectors, from the first, equal to zero, to the last, equal to 65536. Since the sector size is 512 bytes, these storage unit have virtually a maximum capacity of 32 Mbyte.

The input ***WRITE*** allows to select a write access to the device storage, otherwise a read access is intended. The access to the unit is one byte at a time and the output ***DATA_OUT*** should be used for reading, while the input ***DATA_IN*** should be used for writing. The inputs and outputs ***REQ***, ***ACK*** and ***CLR*** operate in a predictable manner, in accordance with what is already described in the case of terminal device (keyboard and screen).

To use the **HDD** device, it is necessary to provide the coordinates of the byte to which the access should be done, writing to the I/O ports 4, 5 and 6, respectively for the storage unit, the field and the byte. Then it is possible to ask a read access (I/O address 2) or write access (I/O address 3). When a read or write operation has been completed, the confirmation signal (acknowledge) is issued from the **HDD** module and sent to the **IRQ** module, as a IRQ2 signal. Anyway, it is not necessary to enable IRQ2 to access the **HDD** module, because reading is always possible, with the difference that if the read data is not yet valid, the value is negative. Similarly, after writing, it is possible to verify the write completion through a read of the same byte: if the value obtained is negative, it means that the operation is not yet complete.

The module **hd** allows to use the virtual storage devices without the need to first

create files: when accessing for the first time, in writing, to an area that had never been used before, the file that represents the sector is created on the fly (the file is created inside the directory where TKGate is working). If it is read a sector that does not exist already, it is produced just a null value (00_{16}) without creating the associated file. The files that are created on writing match the pattern '`hd`***n***`_sector_`***ssssssss*** `.mem`'.

10.15 Macrocode: terminal usage example, through interrupts

«

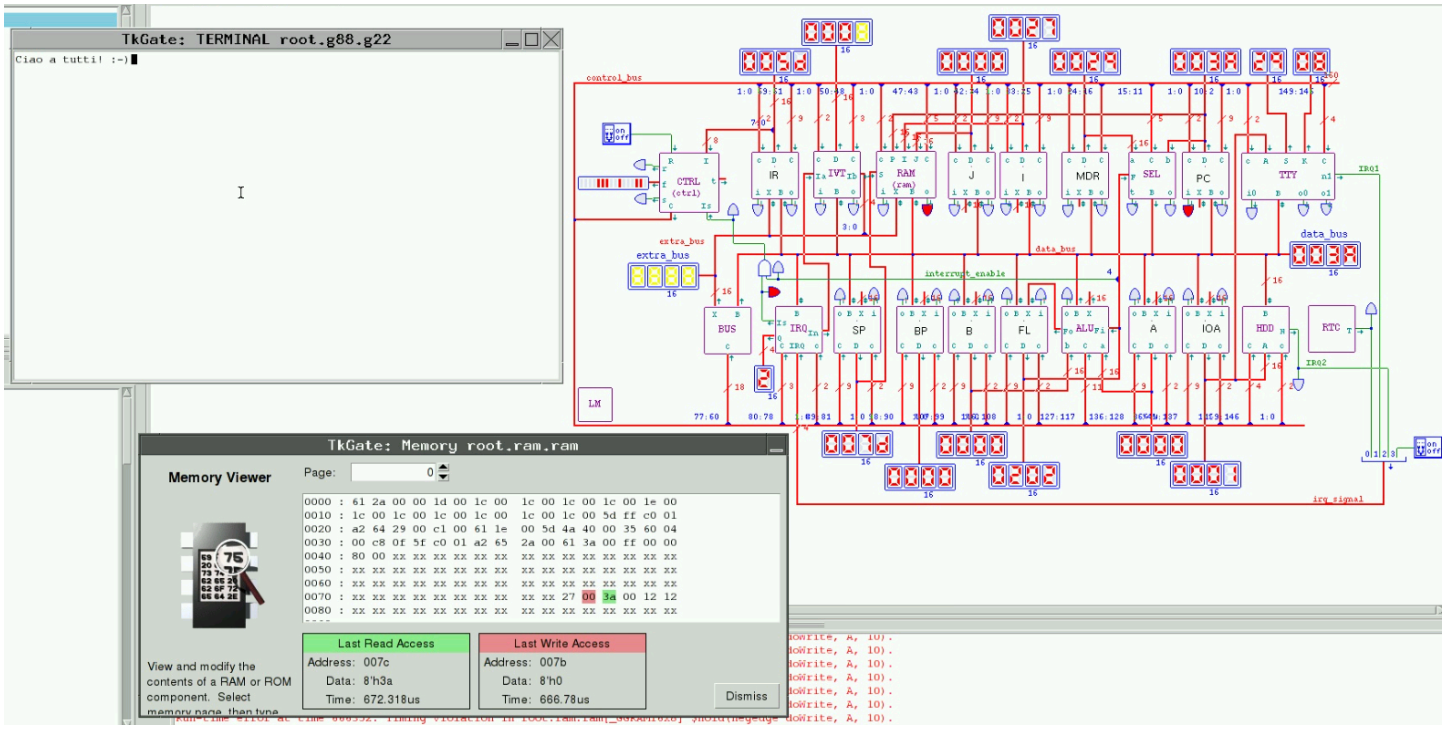
The following code performs the reading of the keyboard, through the interruption generated by the keyboard itself, and the representation of the text typed on the screen. The begin of code defines the location of the IVT table and the associated code for the various table elements.

Listing 10.47. Macrocode managing a terminal through the keyboard interrupts.

```
begin macrocode @ 0
        jump #start
        nop
interrupt_vector_table:
        .short  0x001C  // CPU
        .short  0x001C  // CPU
        .short  0x001C  // CPU
        .short  0x001C  // CPU
        .short  0x001C  // IRQ
        .short  0x001E  // IRQ keyboard
        .short  0x001C  // IRQ
        .short  0x001C  // IRQ
        .short  0x001C  // software
        .short  0x001C  // software
        .short  0x001C  // software
        .short  0x001C  // software
default_interrupt_routine:
        iret
op_code_error:
        stop
keyboard:
        in 1                    // Keyboard read.
        equal
```

```
        jump8z #keyboard_end    // Exit if no data was read.
        out 0                   // Otherwise print on screen the
                                // same value.
        jump #keyboard          // Loop.
keyboard_end:
        iret
start:
        load16 #data_1          // Set the stack location.
        mv %MDR, %SP
        //
        ivtl #interrupt_vector_table
        imrl 0x0F               // All IRQs are enabled.
        seti
keyboard_reset:
        in 1                    // Keyboard read.
        equal
        jump8nz #start          // Loop until the keyboard buffer
                                // is empty.
endless_cycle:
        jump #endless_cycle     // Just loop forever.

stop:                           // Will never reach this point.
        stop
data_0:
        .short 0
data_1:
        .short 0x0080
end
```

Figure 10.48. Keyboard input and screen output. Video: ogv *http://www.youtube.com/watch?v=dgIfZHNTedM*



11.1 General purpose registers 598
11.1.1 Module "RANK" 601
11.1.2 Modules "D_select" and "Q_select" 601
11.2 Module "VR*n*" 602
11.3 Module "RAM" 605
11.4 Module "ALU" and related submodules 608
11.5 Module "BUS" 615
11.6 Module "IRQ" 616
11.7 Modules "D*n*" 617
11.8 Module "IVT" 618
11.8.1 Module "DR*n*" 619
11.9 Module "RTC": real time clock 619
11.10 Module "TTY" 621
11.11 Module "HDD" 625
11.12 Module "CTRL" 630
11.12.1 Modules "F*n*" 633
11.12.2 Module "CLK_CLR" 634
11.13 Memory and microcode fields 636
11.14 Opcodes 641
11.15 Microcode 671
11.16 Macrocode 728

This is another version of the project, with more powerful registers and a better control-unit. There are two data-busses, each one connected with a different ALU

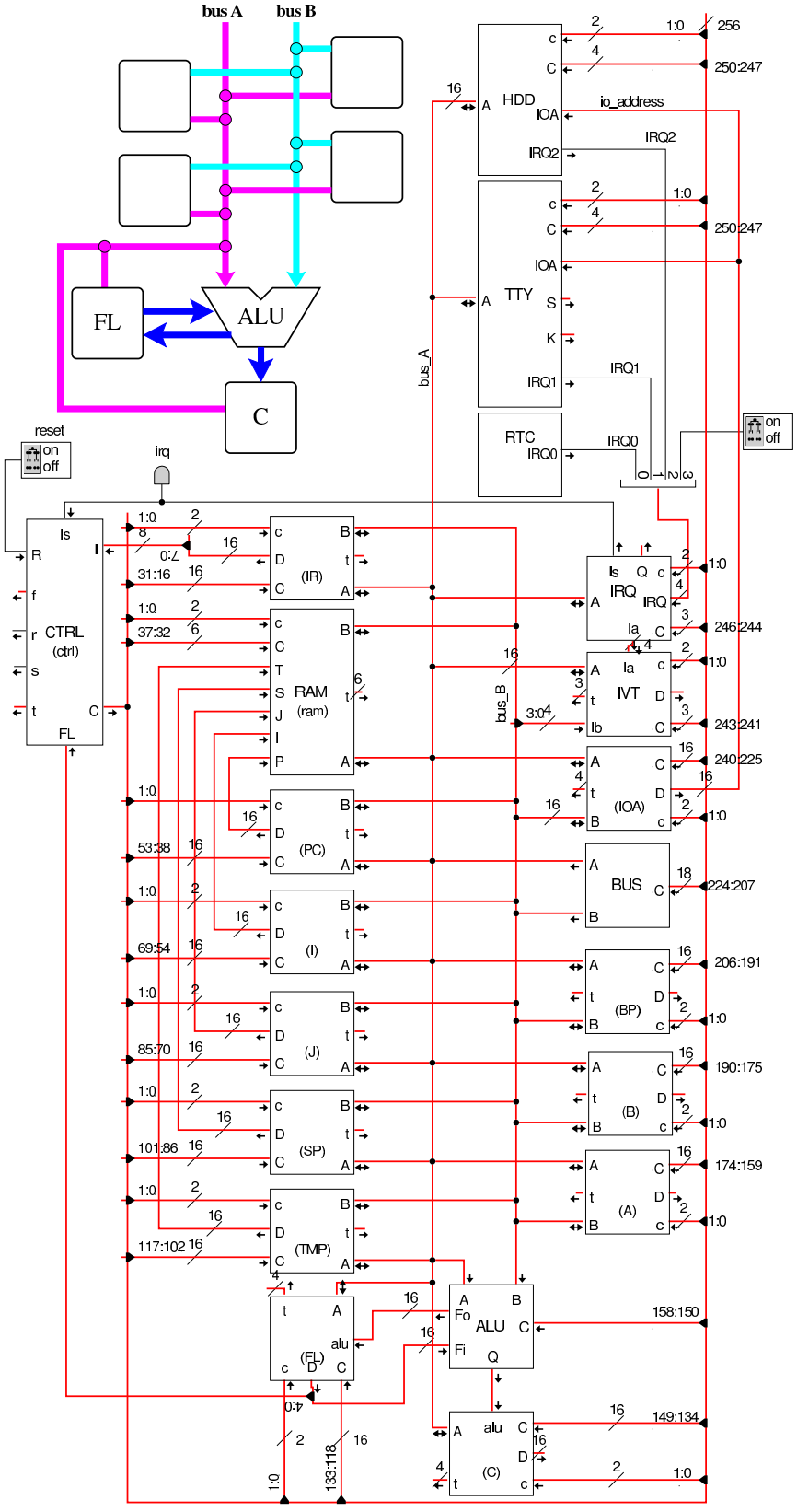
"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

input, but at the moment, the opcode is still reduced to 8-bit and there is no space for enough instructions, so the ALU continues to depend on registers ***A*** and ***B***. The old **SEL** module is not present anymore because the selection work is done inside the control-unit. The ***MDR*** register is not present for this version; instead there is a temporary register ***TMP*** and a new ***C*** register, used as accumulator.

This version does not access the memory correctly, as some previous version do; this fact is evidenced by a lot of warning produced by the TKGate simulation. The next version "M" has a different memory management, with a new ***MDR*** register, but it is very slow compared with the current version.

Attachments	Description
attachments/xcpu/xcpu-d.v	TKGate Verilog netlist source file.
attachments/xcpu/xcpu-d.gm	TKGate microcode and macrocode source file.
attachments/xcpu/xcpu-d-terminal.vpd.tcl	TCL/Tk script related to the terminal module **TTY**.

Figure 11.2. Simple CPU, version “D”: two data-busses and an accumulator register.



11.1 General purpose registers

The registers are made in a similar way as the previous version, but internally there are new modules that can be incremented or decremented without restrictions, before or after reading the register. There are two types of registers: the following figure shows the two external layouts.

Figure 11.3. 16-bit registers: external appearance.

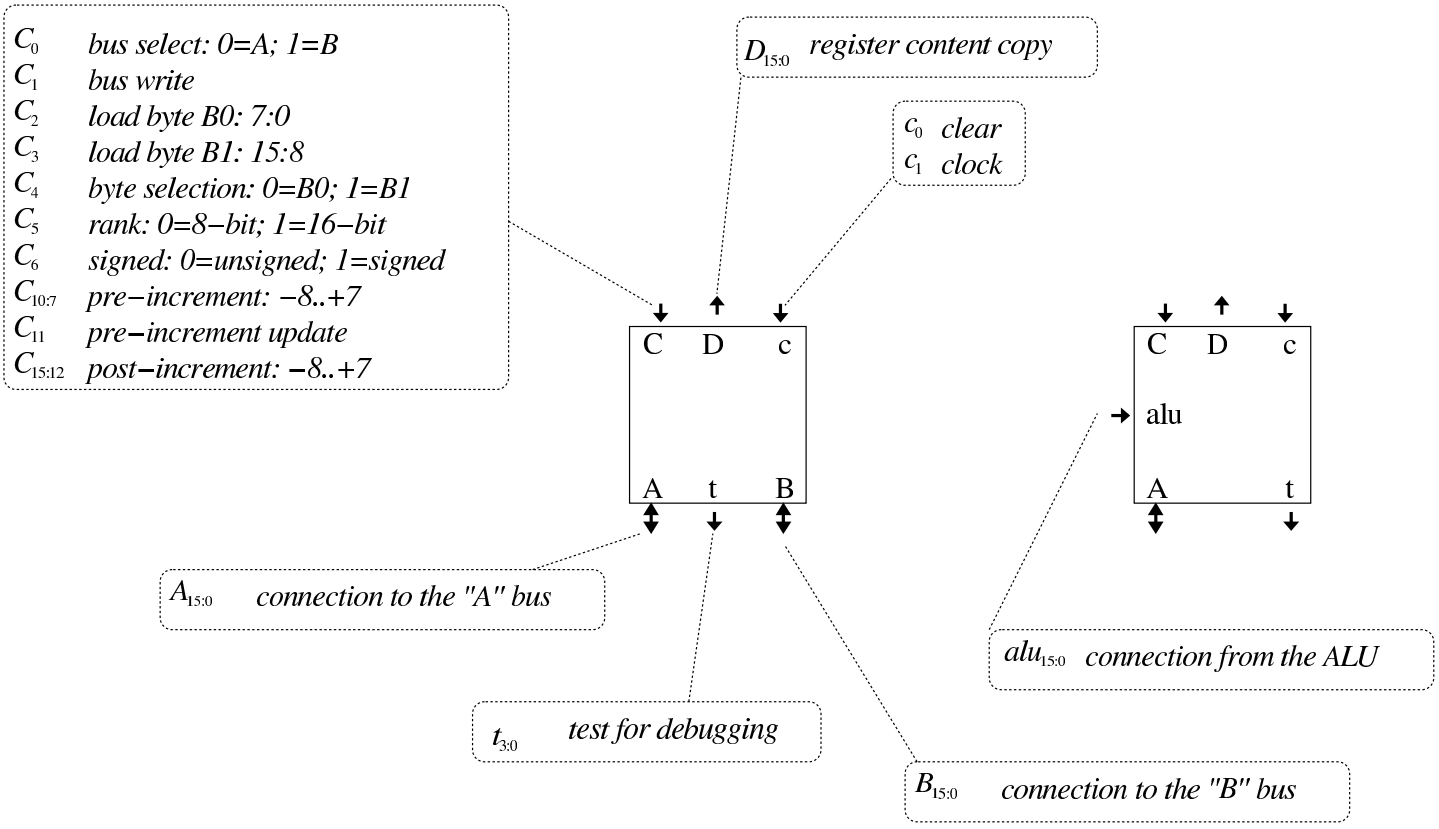
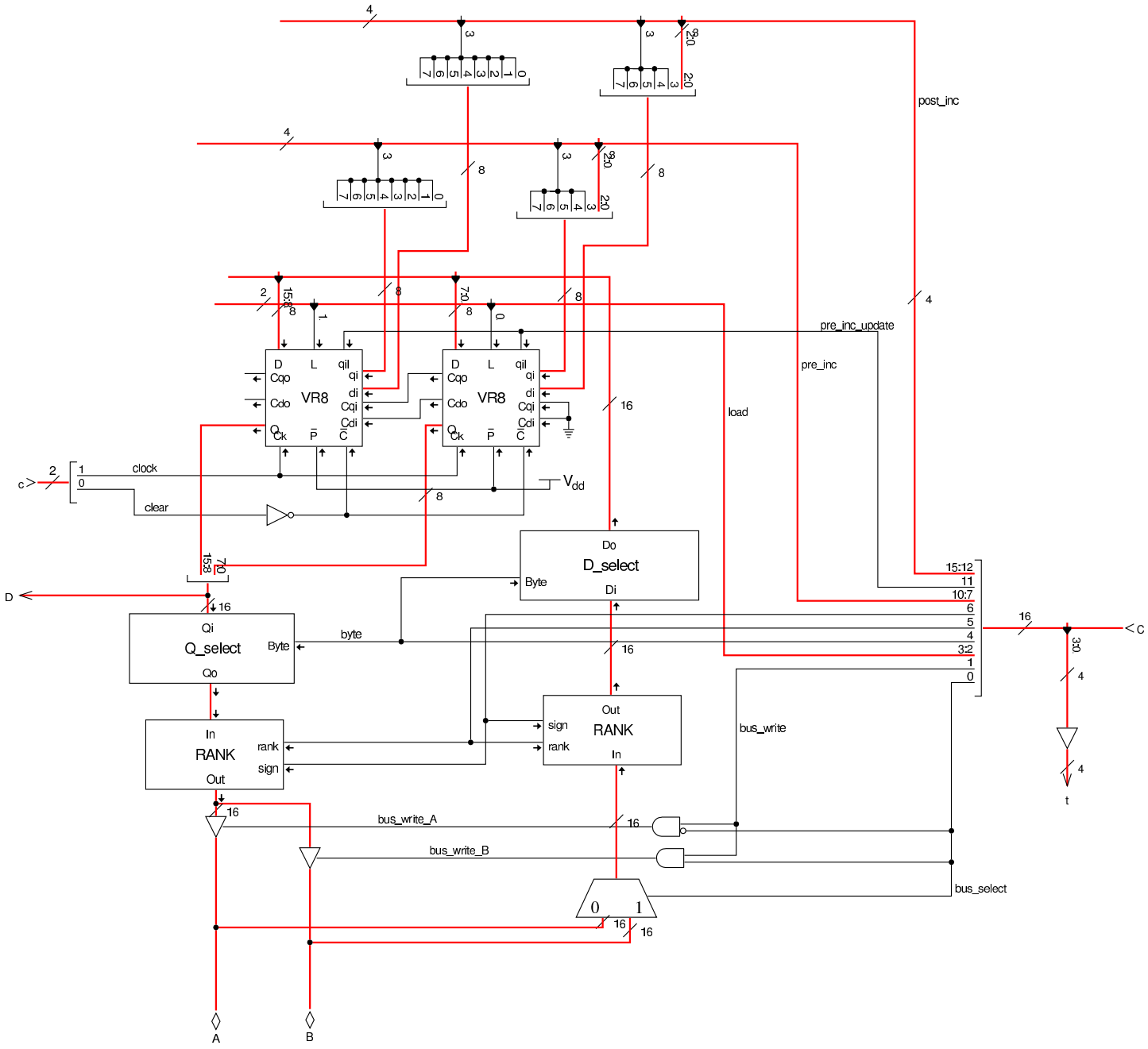
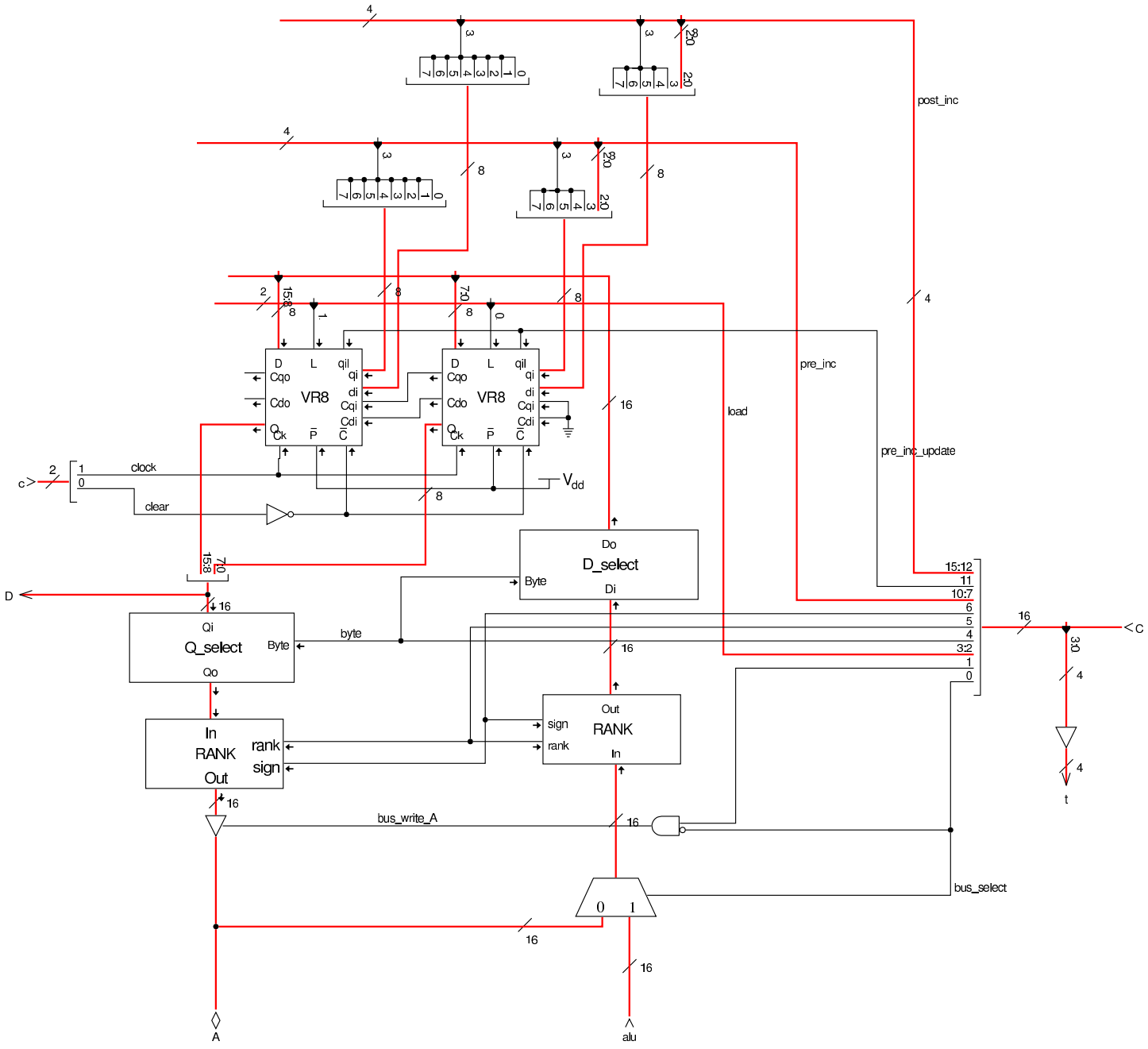


Figure 11.4. 16-bit registers: internal structure.



The ***C*** and ***FL*** registers are made with a slightly different module, where the "B" bus connection is named ***alu*** and is only able to receive data.

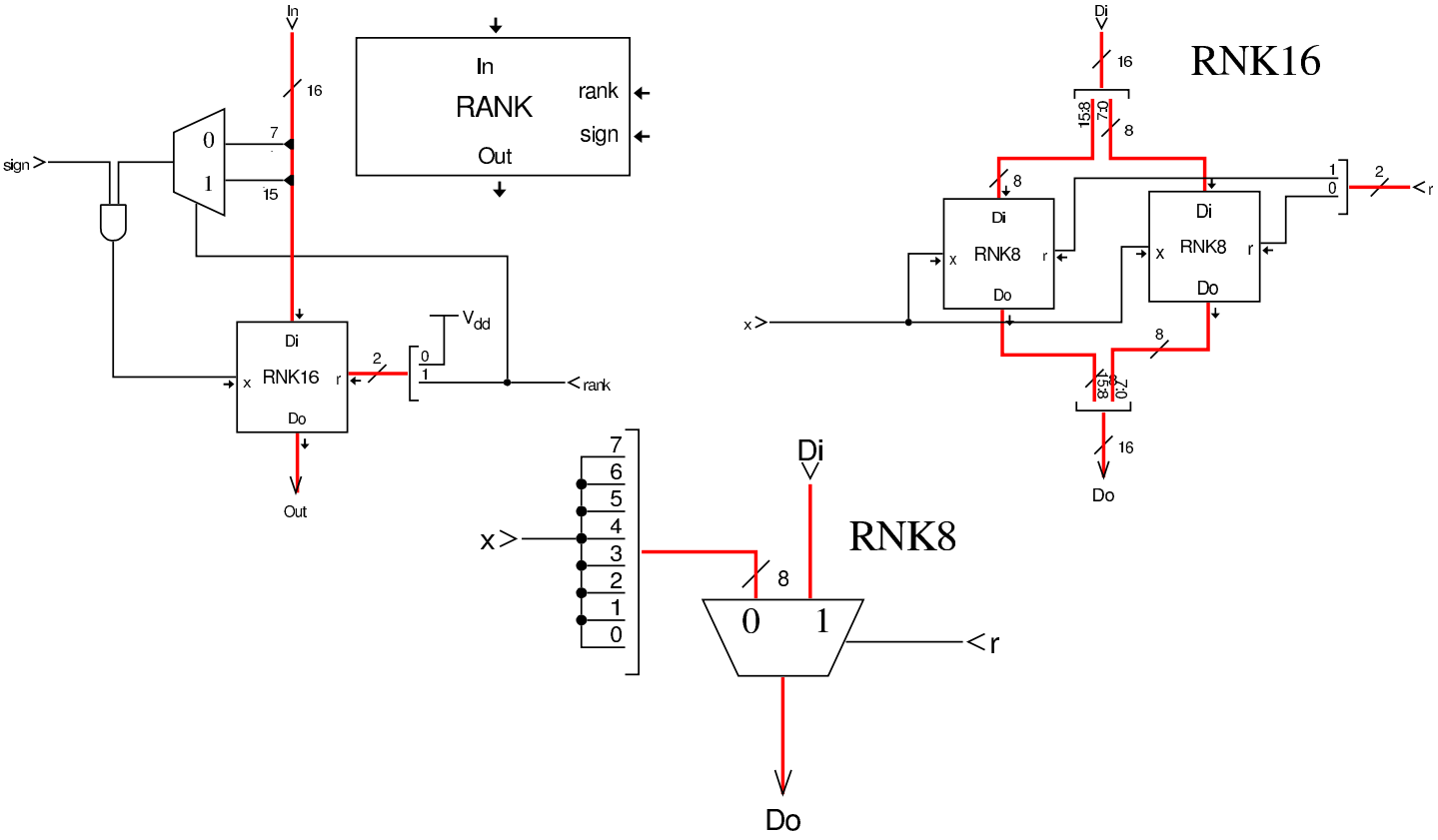
Figure 11.5. Alternative 16-bit registers structure.



The control line C_4 (*byte selection*) should be zero when C_5 (*rank*) is equal to one (16-bit); otherwise, it is used to select the byte: when writing to the bus, the selected byte from the register is written to the bus (to the lower bus side); when reading from the bus, the lower byte from the bus is stored to the selected register byte. The following sections show the modules responsible for the rank reduction and the byte selection.

11.1.1 Module "RANK"

Figure 11.6. Rank reduction module: the module **RANK** and its submodules.

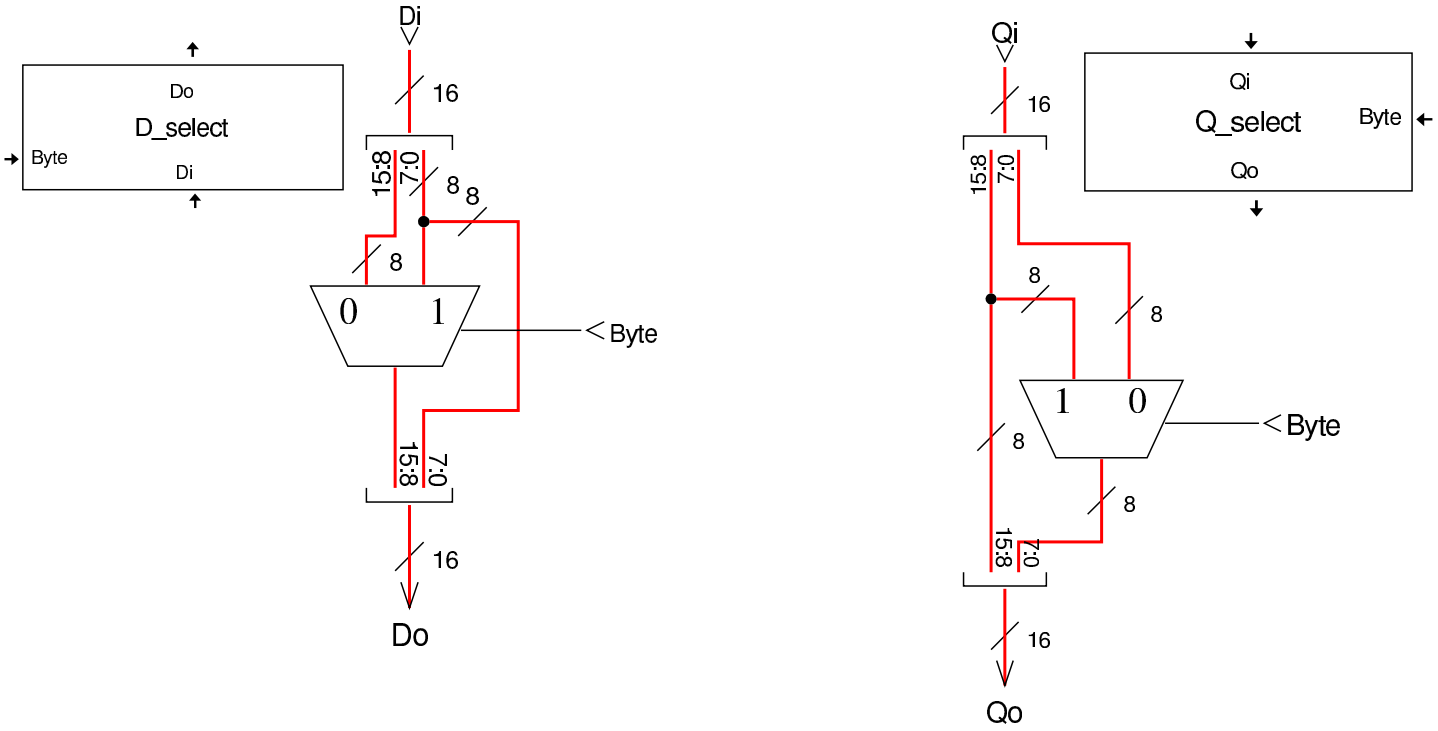


11.1.2 Modules "D_select" and "Q_select"

The incoming byte selection module (**D_select**), when ***Byte*** is equal to zero, does not change the output, so that ***Do*** is equal to ***Di***. When ***Byte*** is equal to one, the low input byte is replicated to the high output byte; that is: ***Do*** is equal to $Di_{7:0} \cdot 2^8 + Di_{7:0}$. This alteration is used only to copy the byte received from the bus to the selected position inside the register.

The outgoing byte selection (module **Q_select**) is used to select from the register a byte to be written to the data-bus. When ***Byte*** is equal to zero, there is no change to the output. When ***Byte*** is equal to one, the byte $Di_{15:8}$ is copied to the byte $Do_{7:0}$ ($Do = Di_{15:8} \cdot 2^8 + Di_{15:8}$). The module **RANK** does the rest of the work, cleaning the most significant byte or setting it if the value is negative and is to be treated as a signed value.

Figure 11.7. Left side: incoming byte selection. Right side: outgoing byte selection.

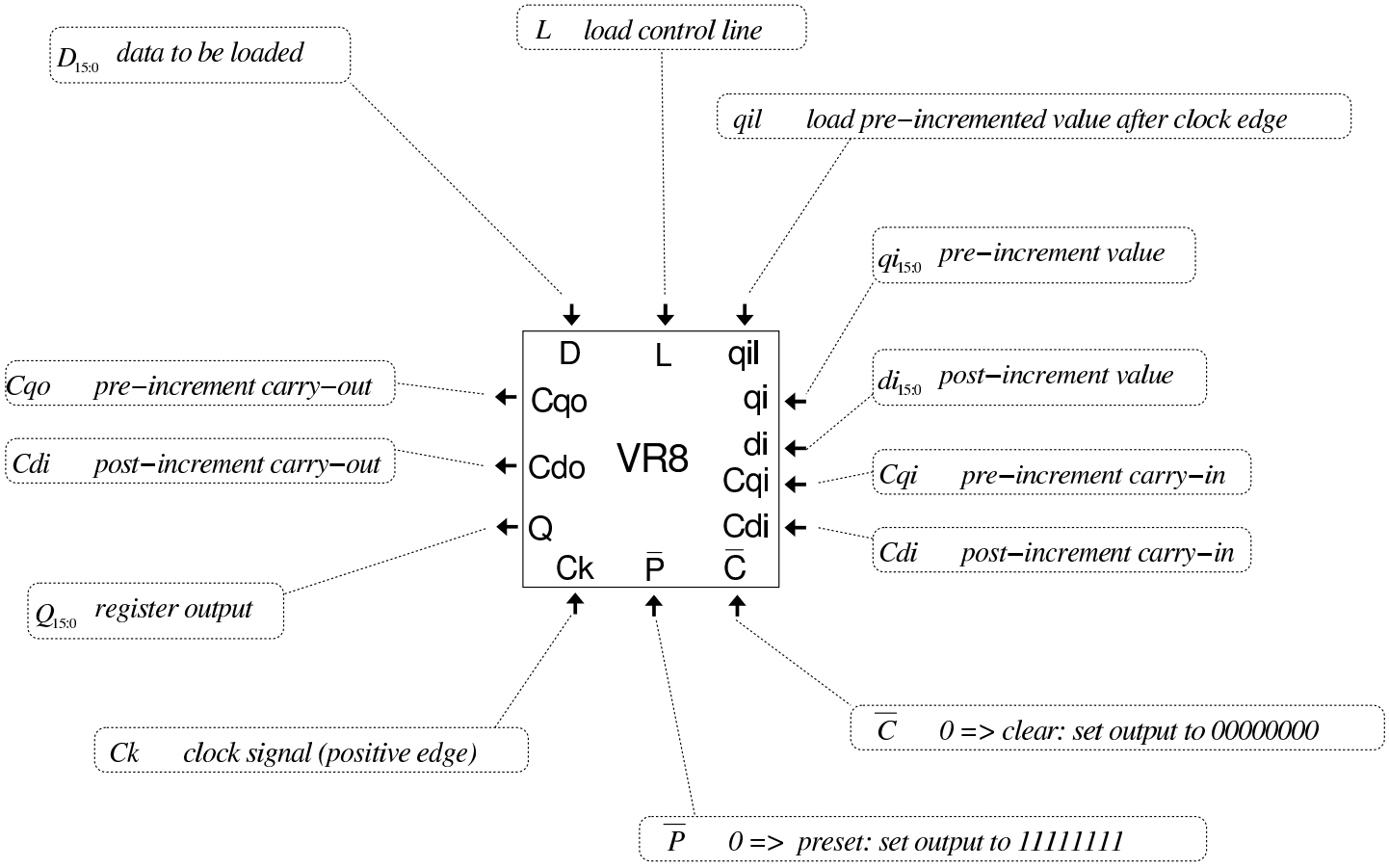


The most important module inside the general purpose register is **VR8** (*versatile register*), which is a 8-bit data register, shown inside the figures of the following section.

11.2 Module "VR*n*"

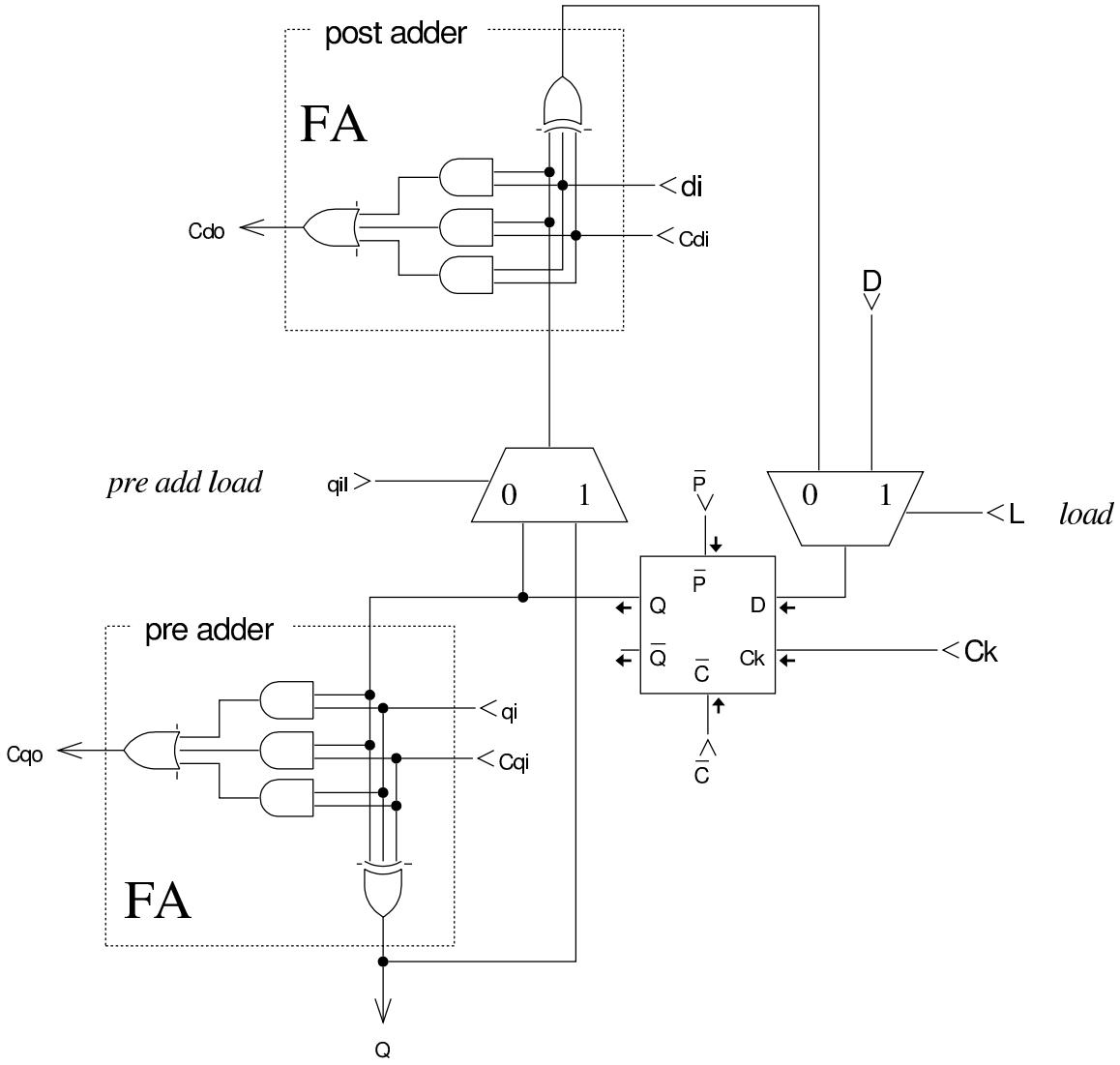
The module **VR*n*** is ***n***-bit register that allows to add a value, positive or negative, before reading the register or after reading it. In particular, if the value is added before reading, the register might not be updated with it, when the clock signal reaches the edge.

Figure 11.8. Module ***VR8***: how it works.



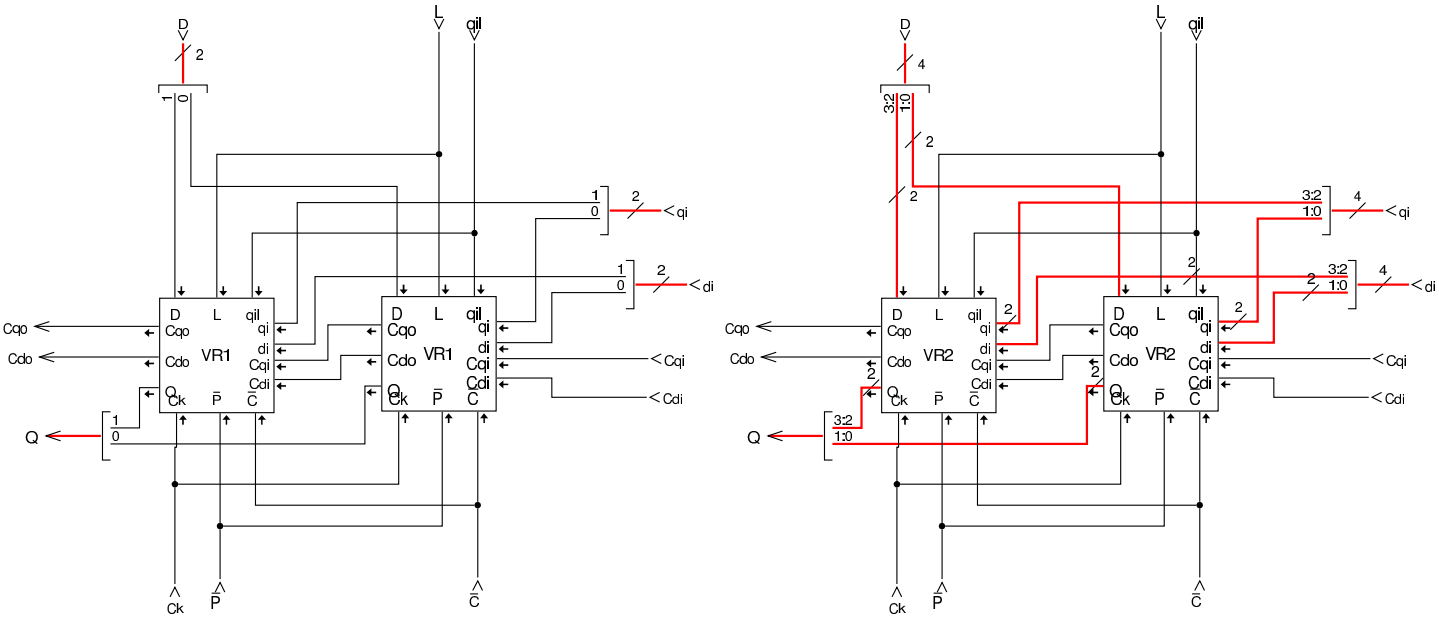
It all starts from module **`VR1`** which is made of a D flip-flop and two full adders, as shown in the following figure.

Figure 11.9. Module **VR1**.



Two modules **VR1** make a module **VR2**; two modules **VR2** make a module **VR4** and so on.

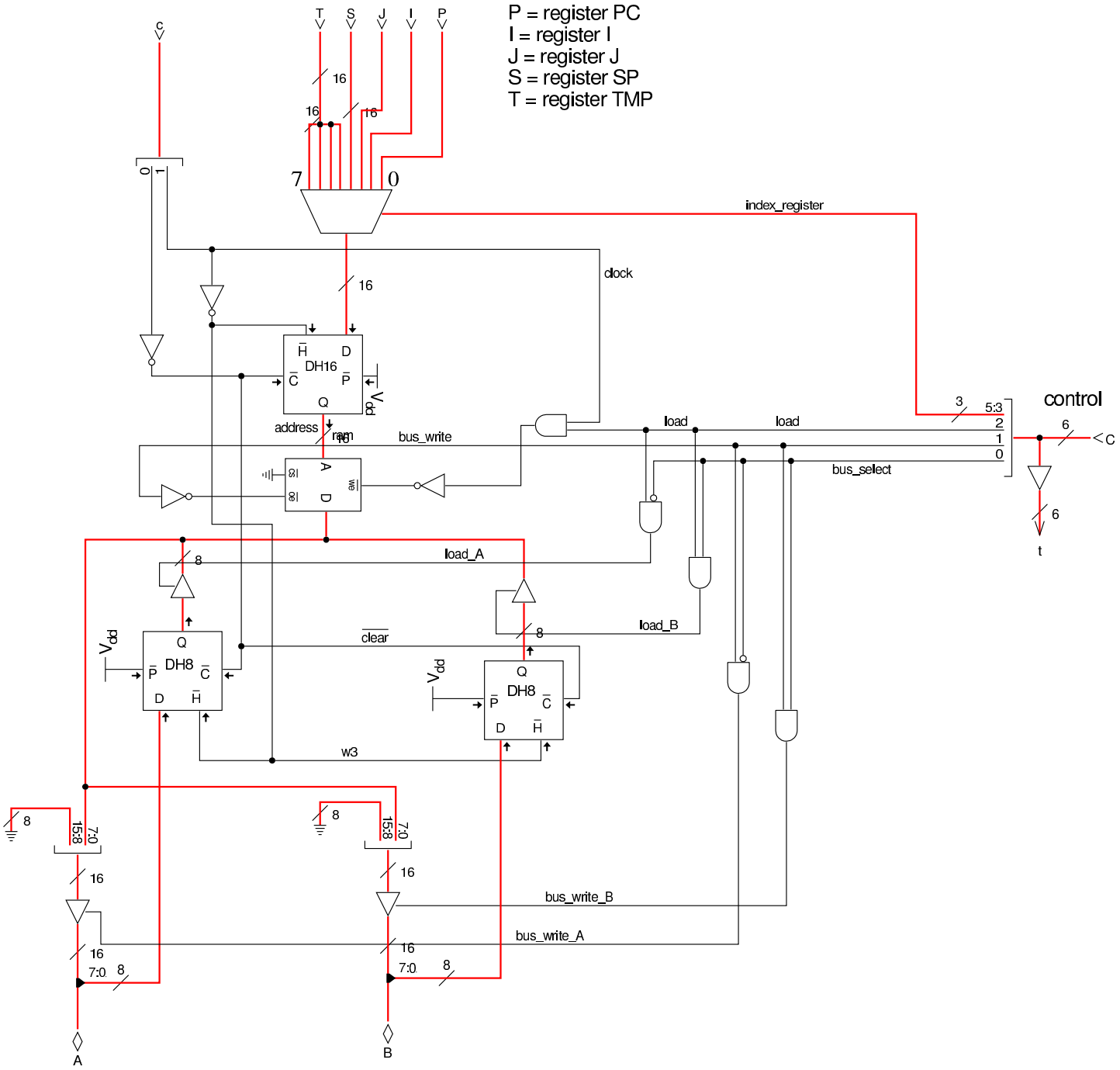
Figure 11.10. Modules **VR2** and **VR4**.



11.3 Module "RAM"

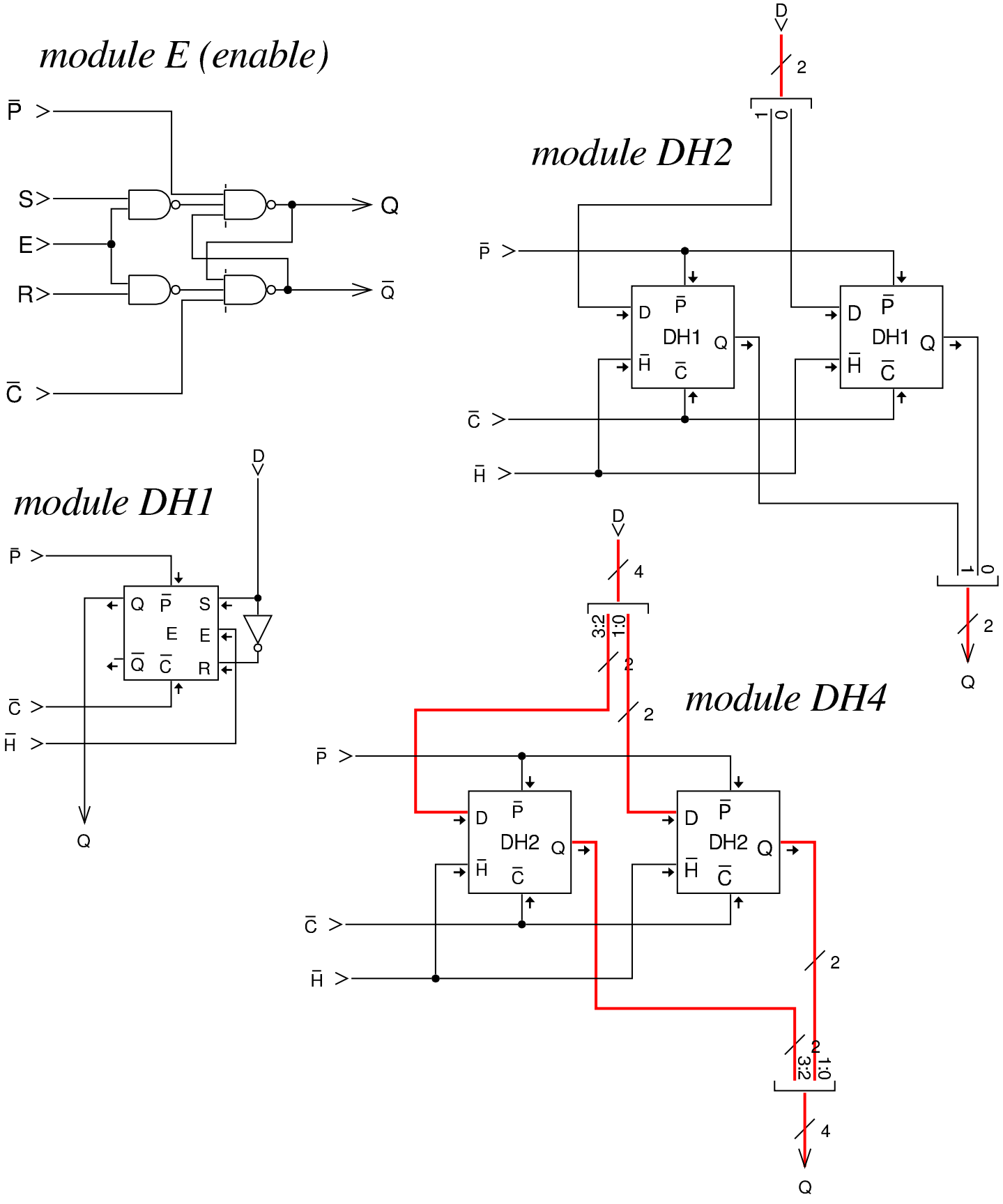
The module **RAM** is similar to the previous version of the project. In particular, there is now the ability to receive an address from the ***TMP*** register and, when the memory is written, there are two added D latches (modules **DH8**) which hold the data read from the bus.

Figure 11.11. The **RAM** module.



Modules **DH*n*** are the same as the previous version, starting from a controlled NAND SR latch, which becomes a D latch, as the following figure shows.

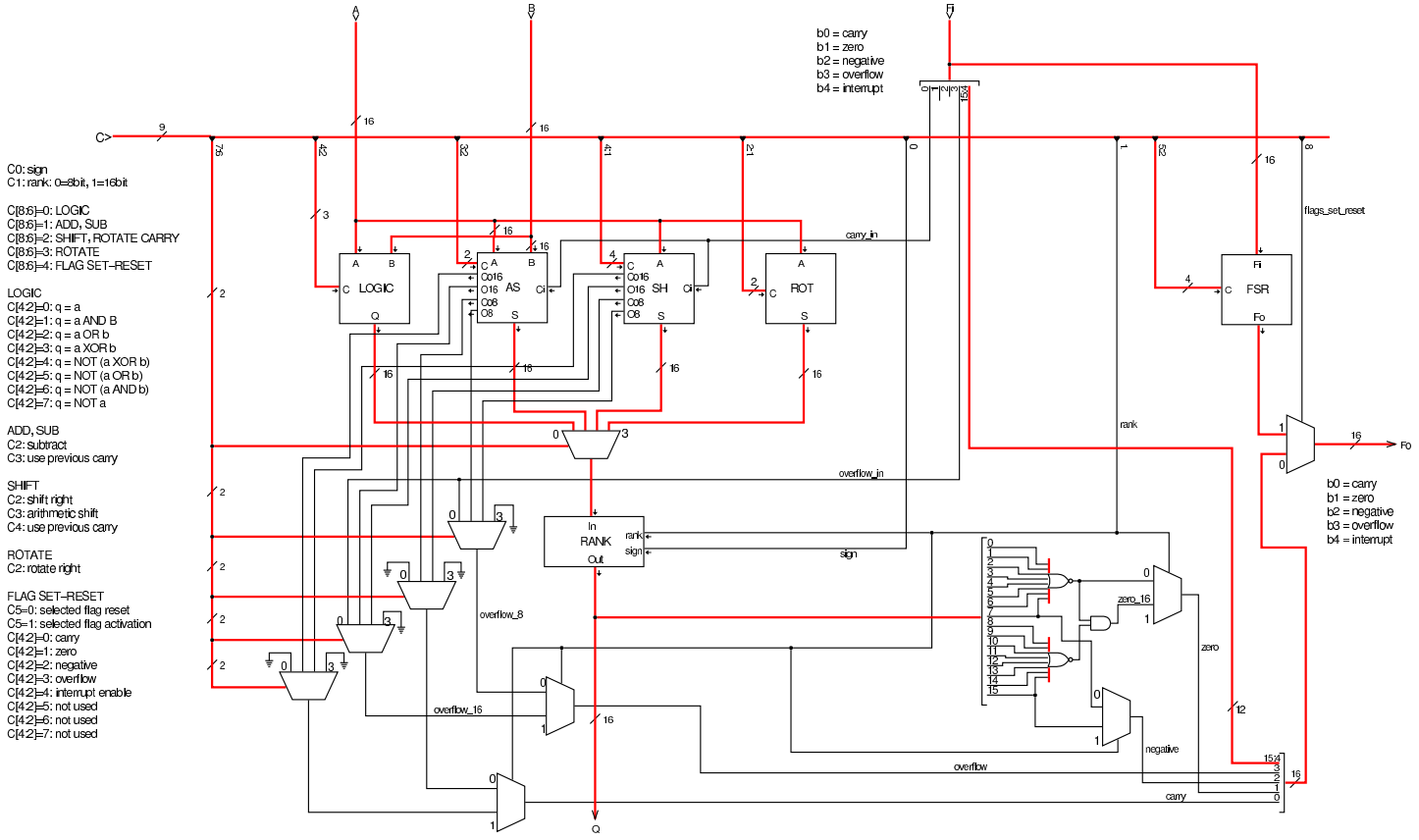
Figure 11.12. The steps from the SR latch to the **DH4** module.



11.4 Module "ALU" and related submodules

The module **ALU**, is similar to the previous version; the most important difference is that there is a single set of ALU status flags: *carry*, *zero*, *negative*, *overflow*. The **ALU** module contains the same module **RANK** already seen inside the general purpose registers.

Figure 11.13. Module **ALU**.



The other components inside the **ALU** module are just the same as the previous version.

Figure 11.14. Module **LOGIC**: the logic unit.

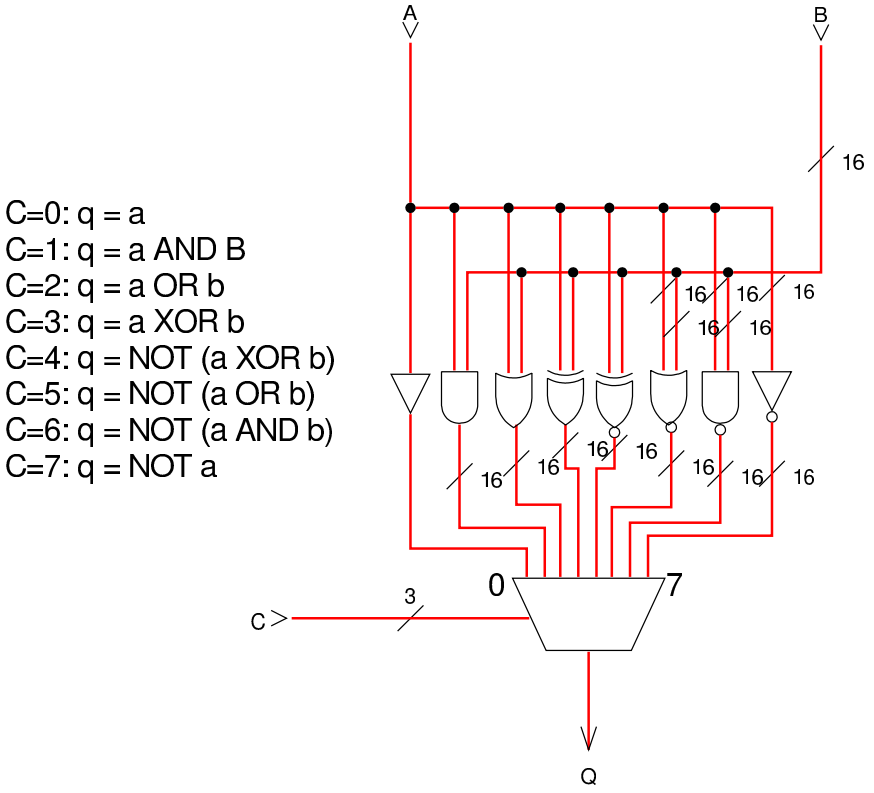


Figure 11.15. Module **AS**: addition and subtraction

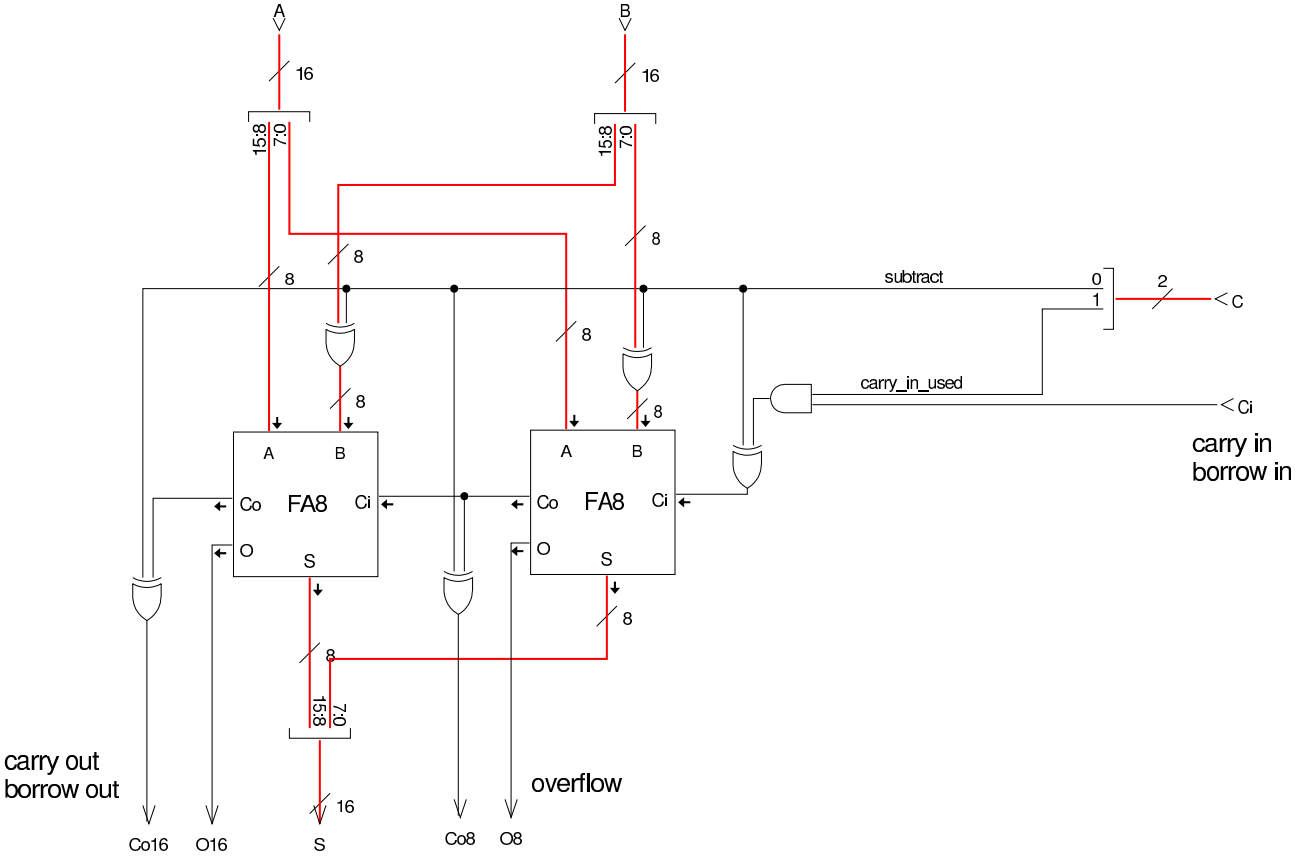


Figure 11.16. Modules **FA*n***: ***n***-bit full adder.

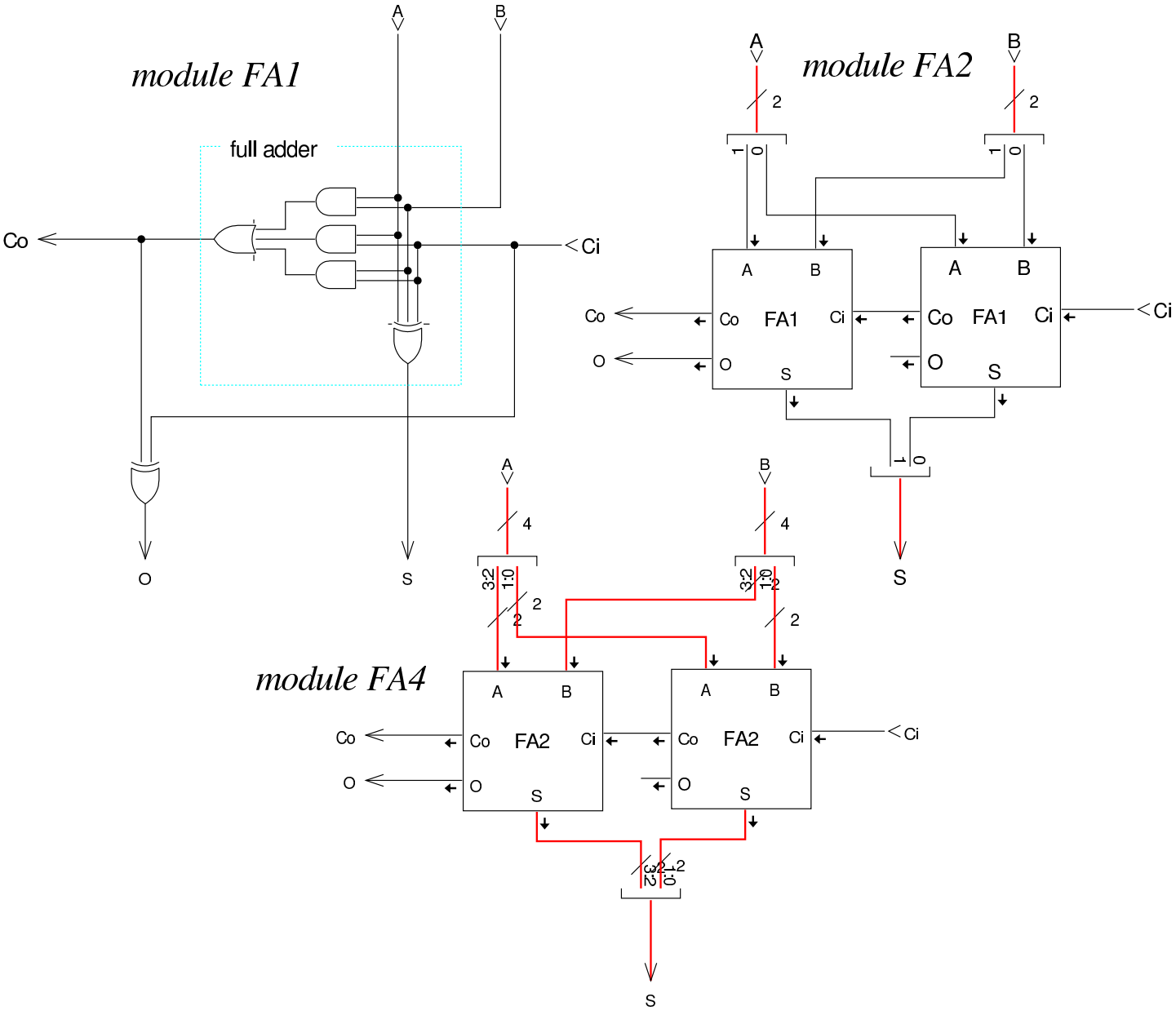


Figure 11.17. Module **SH**: bit shift.

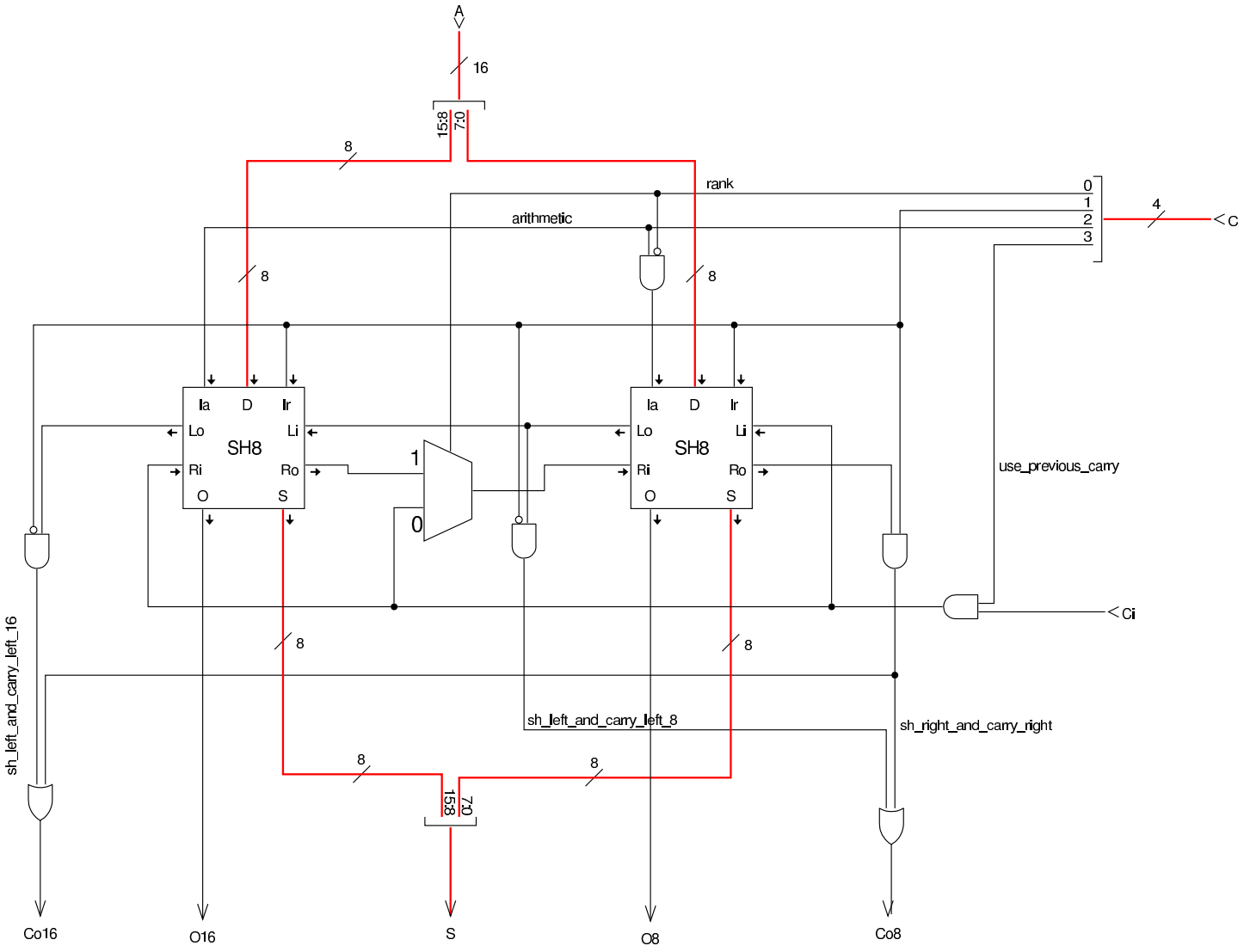
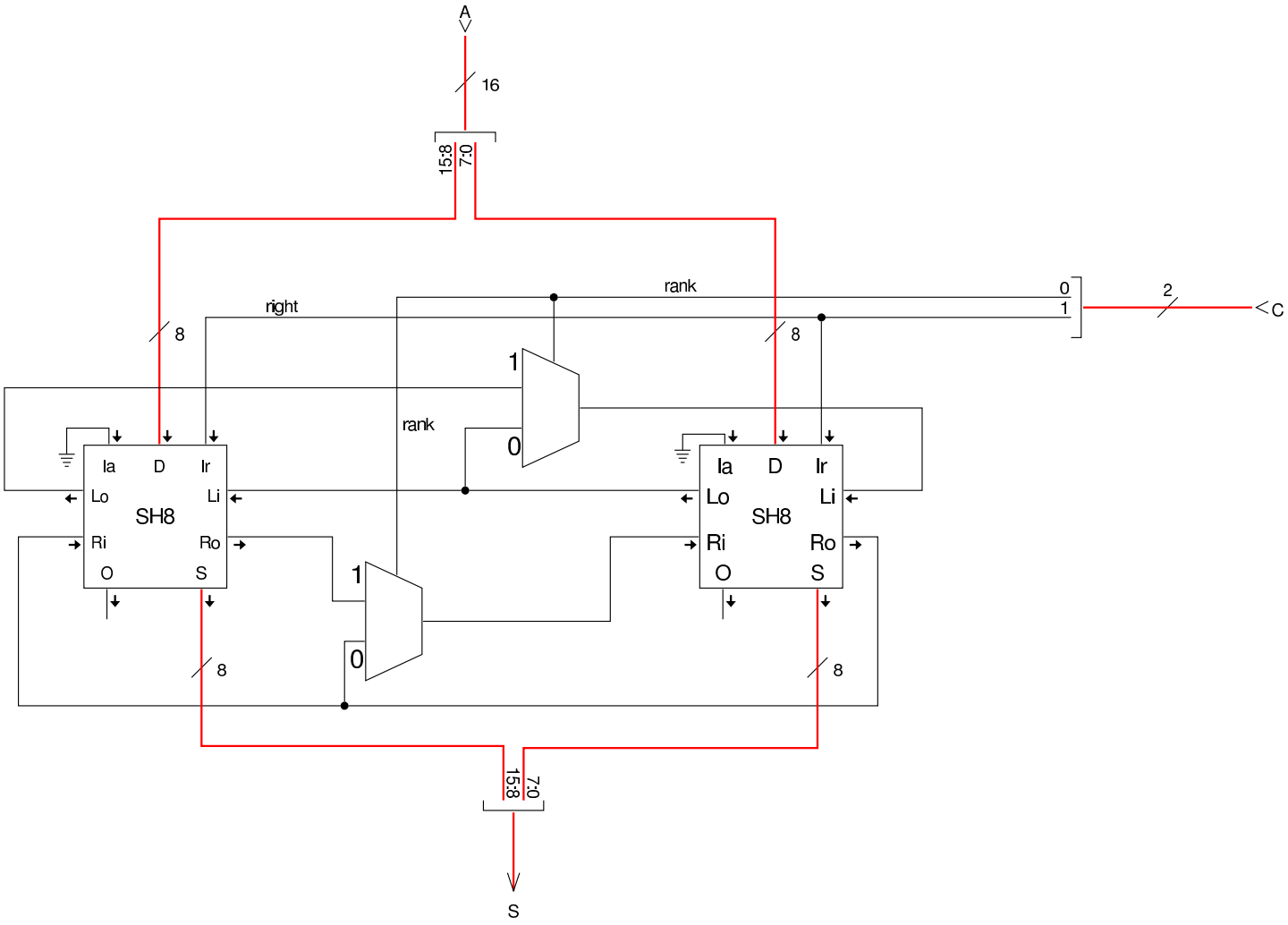
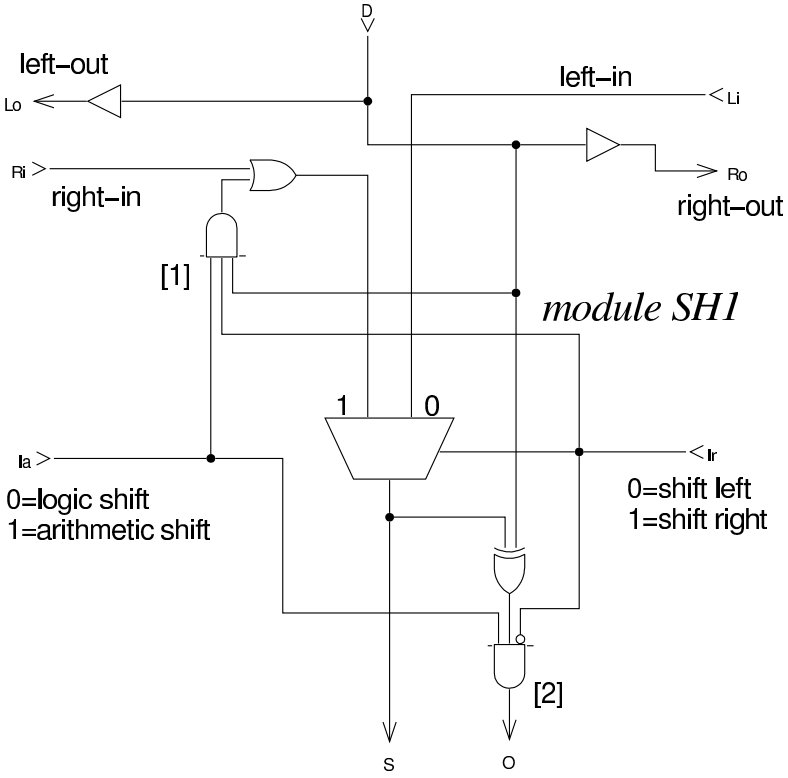


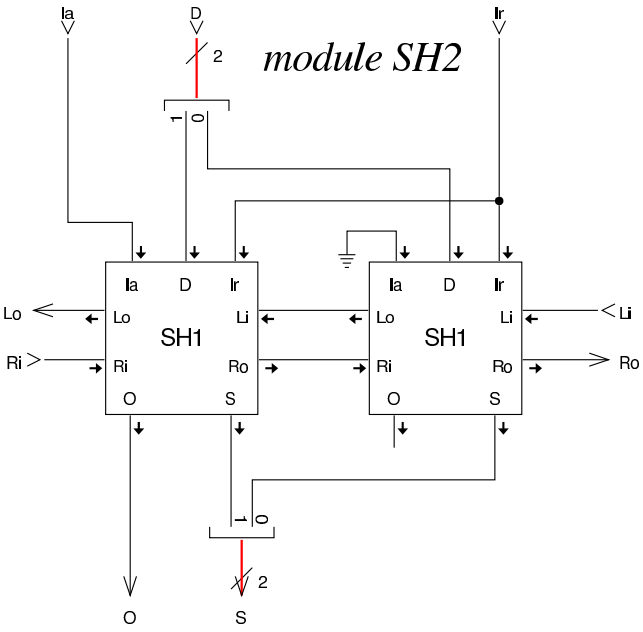
Figure 11.18. Module **ROT**: bit rotation.

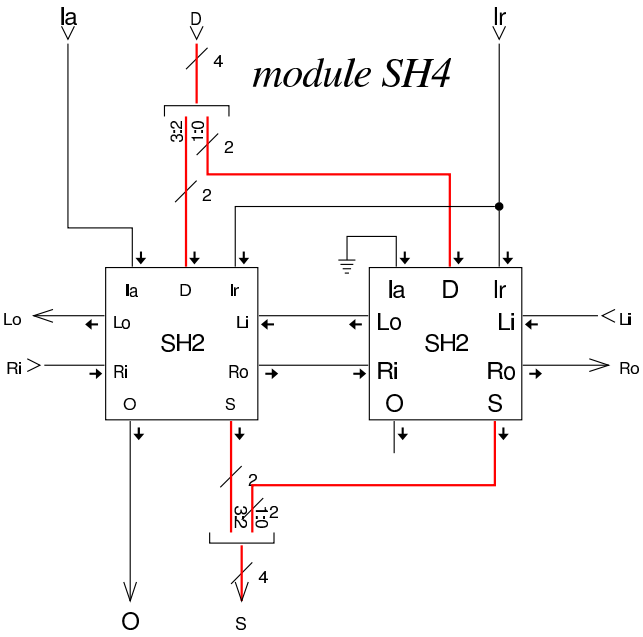


The modules **SH*n*** are one-bit logic or arithmetic shifter. It all starts from module **SH1**, as the following figure shows.

Figure 11.19. Module **SH*n***: one-bit logic or arithmetic shift.



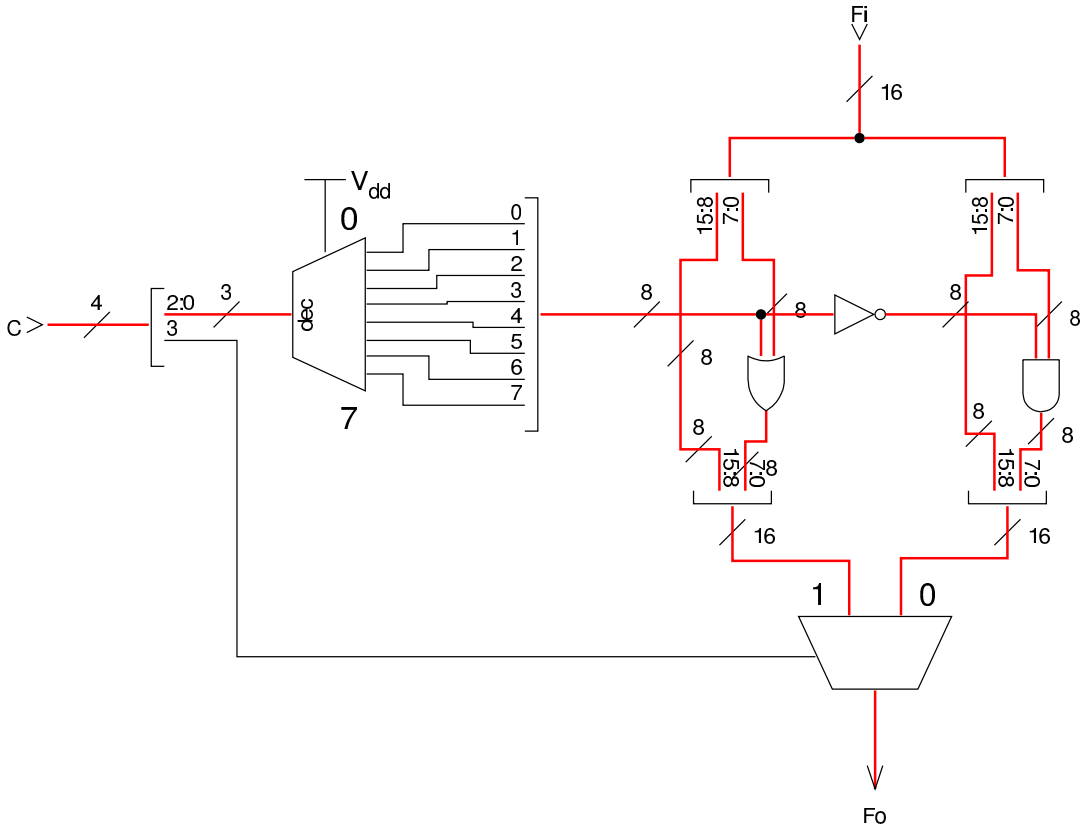




[1] Arithmetic shift conditions:
- the requested shift is arithmetic;
- the requested shift is to the right;
- the most significant bit value is one.

[2] Overflow conditions:
- the requested shift is arithmetic;
- the requested shift is to the left;
- the most significant bit value is changed.

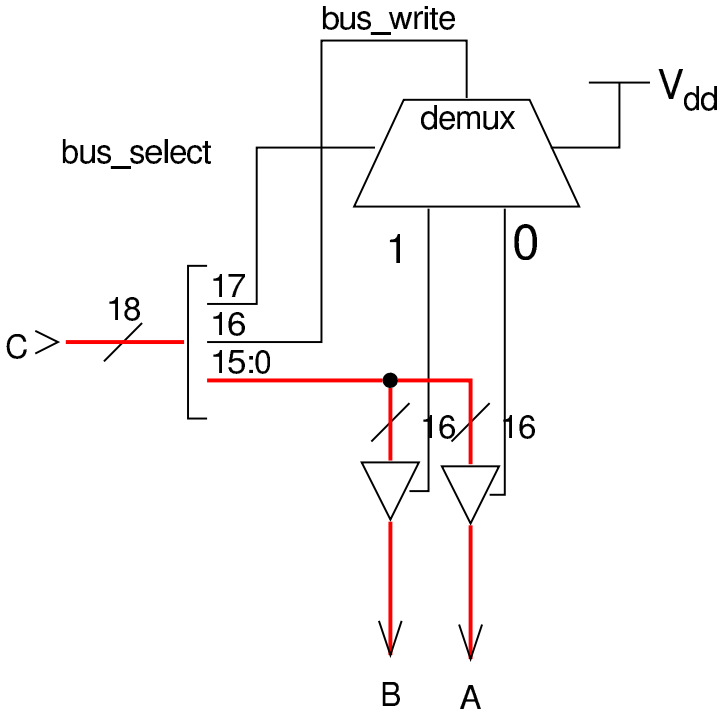
Figure 11.20. Module **FSR**: flags set-reset.



11.5 Module "BUS"

The module **BUS** is used to let the control unit to put a value inside the data-bus. This module **BUS** is almost the same as the previous version: only the connection names to the bus are changed and the control line used to select the destination bus.

Figure 11.21. Module **BUS**.

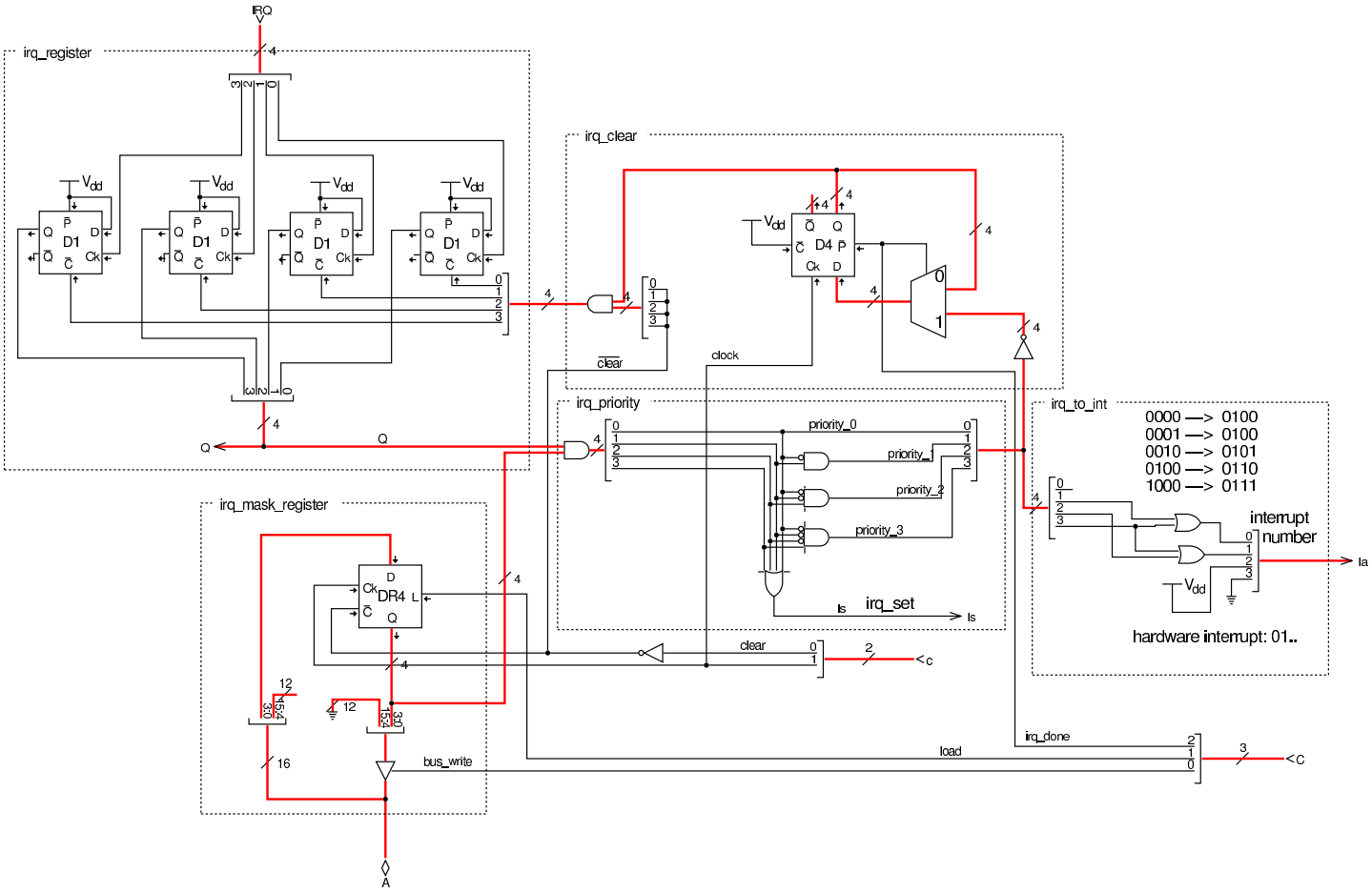


11.6 Module "IRQ"

«

The **IRQ** module is just the same as the previous version, but here is connected to the A data-bus. All the details about the module functionality should be found at the previous version description.

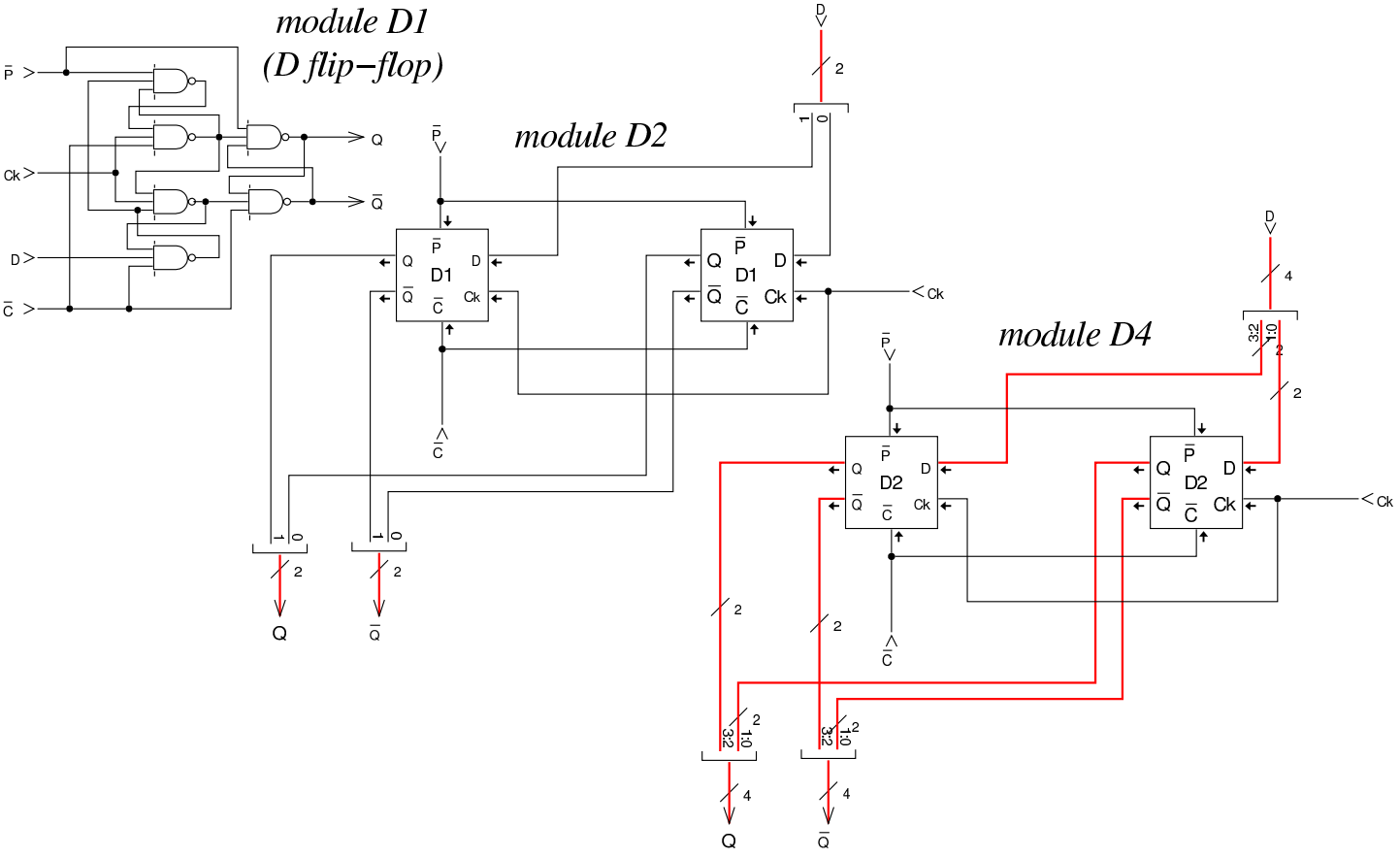
Figure 11.22. Module **IRQ**.



11.7 Modules "D*n*"

The modules **D*n*** are D flip-flops, in paralel, as the following figure shows.

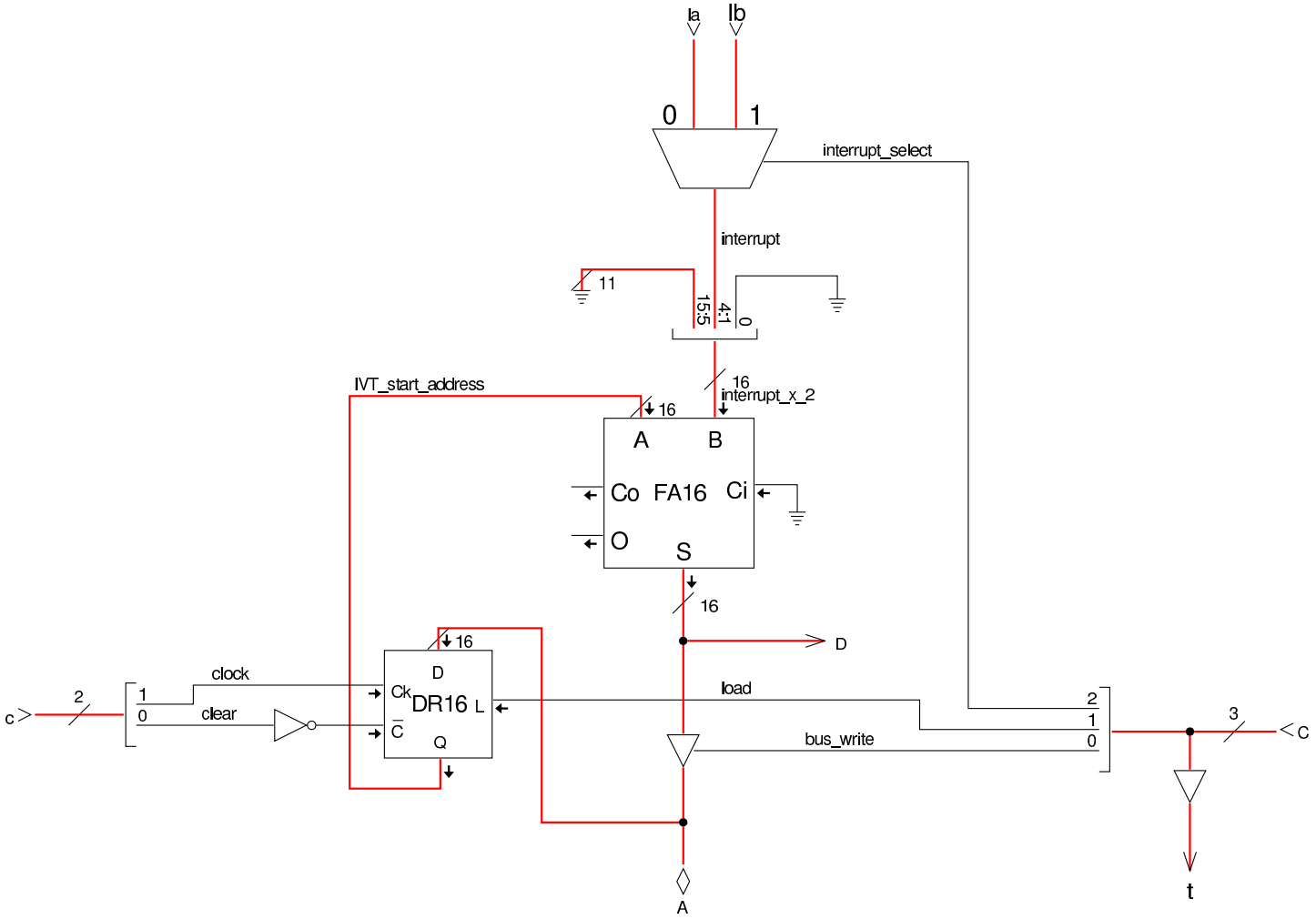
Figure 11.23. Building **D*n*** modules.



11.8 Module "IVT"

The **IVT** module is just the same as the previous version, but here is connected to the A data-bus. All the details about the module functionality should be found at the previous version description.

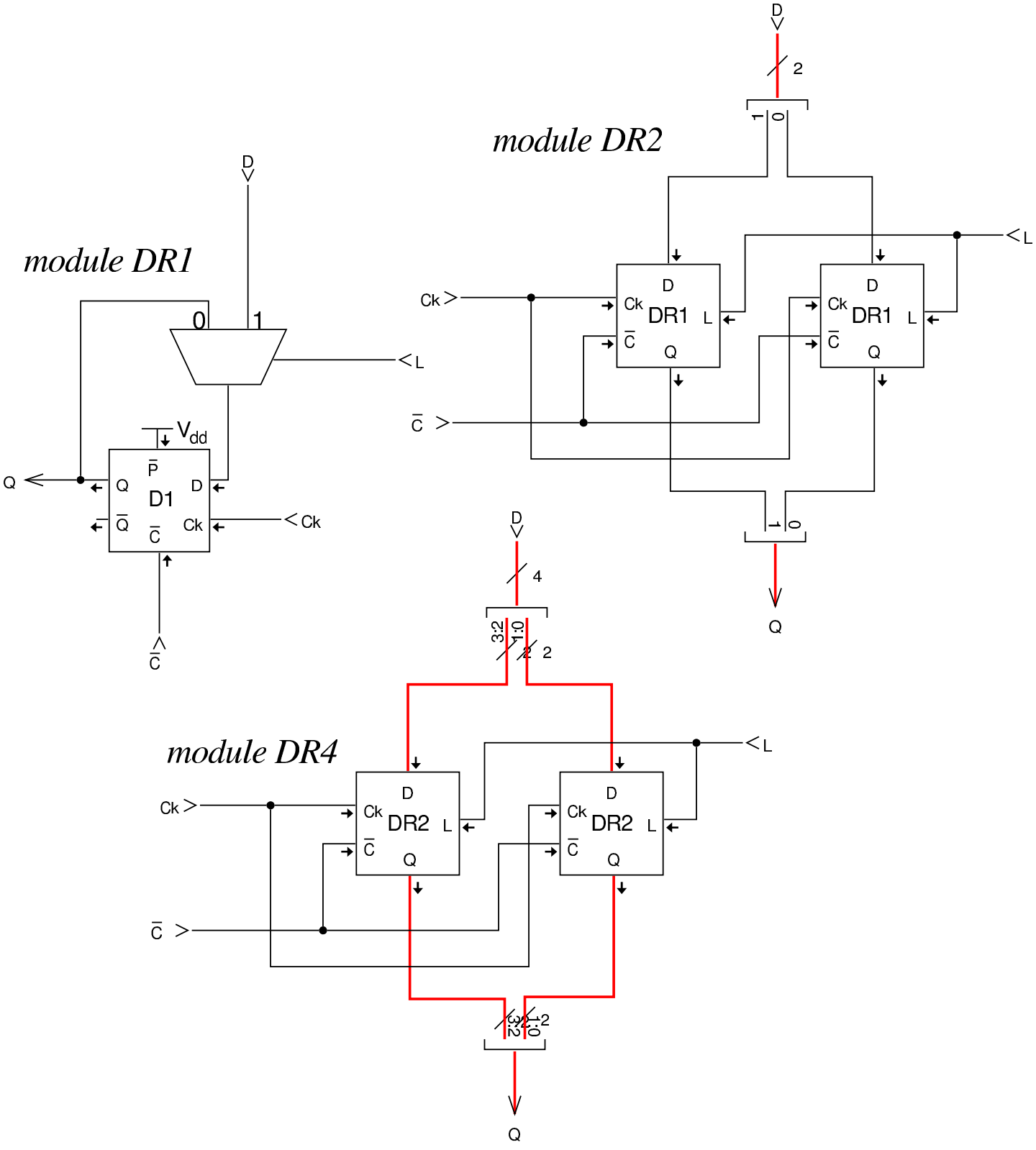
Figure 11.24. Module **IVT**.



11.8.1 Module "DR*n*"

The modules **IRQ** and **IVT** use modules **DR*n***, which are ***n***-bit data registers.

Figure 11.25. Building **DR*n*** modules.



11.9 Module "RTC": real time clock

«

The **RTC** module (*real time clock*) is just the same as the previous version: it generates a 1 Hz impulse, producing the hardware interrupt IRQ0. All the details about the module functionality should be found at the previous version description.

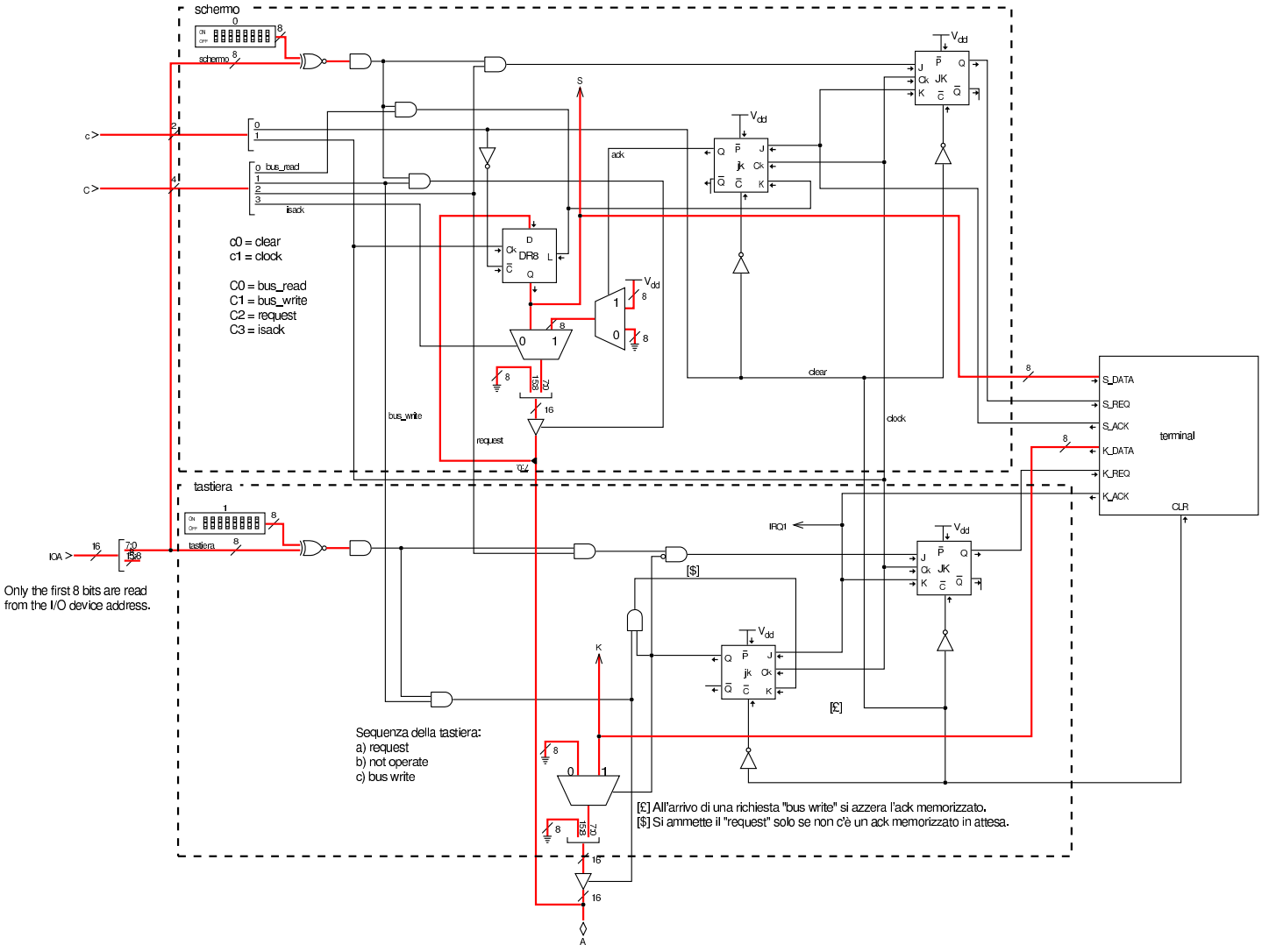
Listing 11.26. Module **RTC**: Verilog declaration.

```
module RTC (T);
  output T;
  reg p;
  always
    begin
      p = 0;
      $tkg$wait (500);
      p = 1;
      $tkg$wait (500);
    end
  assign T = p;
endmodule
```

11.10 Module "TTY"

The **TTY** module is just the same as the previous version: it is a textual screen-keyboard terminal interface. All the details about the module functionality should be found at the previous version description.

Figure 11.27. Module **TTY**.



Listing 11.28. Module **terminal**: Verilog code.

```
module terminal(K_DATA, K_REQ, K_ACK, S_DATA, S_REQ, S_ACK, CLR);

output K_ACK;
output S_ACK;
output [7:0] K_DATA;
input [7:0] S_DATA;
input K_REQ;
```

```
input S_REQ;
input CLR;
reg k_ready;
reg [7:0] key;
reg s_ready;

  initial
    begin
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  always
    begin
      @(posedge CLR)
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  initial $tkg$post("TERMINAL","%m");

  always
    begin
      @ (posedge K_REQ);
      # 5;
      key = $tkg$recv("%m.KD");
      # 5;
      k_ready = 1'b1;
      # 5;
      @ (negedge K_REQ);
      # 5;
      k_ready = 1'b0;
    end

  always
```

```
    begin
      @(posedge S_REQ);
      # 5;
      $tkg$send("%m.SD",S_DATA);
      # 5;
      s_ready = 1'b1;
      # 5;
      @(negedge S_REQ);
      # 5;
      s_ready = 1'b0;
    end

  assign S_ACK = s_ready;
  assign K_DATA = key;
  assign K_ACK = k_ready;

endmodule
```

Listing 11.29. File ‘`share/tkgate/vpd/terminal.tcl`’ for TCL interface.

```
image create bitmap txtcurs -file "$bd/txtcurs.b"

VPD::register TERMINAL
VPD::allow TERMINAL::post
VPD::allow TERMINAL::data

namespace eval TERMINAL {
    # Public variables declarations: the variables $terminal_...
    # are arrays of which only the element $n is used;
    # that element identifies uniquely the working interface instance.
    variable terminal_w
    variable terminal_pos
    #
    variable KD
    # Function requested by TKGate to create the interface.
    proc post {n} {
        variable terminal_w
        variable terminal_pos
        # Create the window and save the object element in a $terminal_w array element.
        set terminal_w($n) [VPD::createWindow "TERMINAL $n" -shutdowncommand "TERMINAL::unpost $n"]
        # For convenience, copy the object reference inside the local
        # variable $w; then, the variable $w will be used as a reference to the object.
        set w $terminal_w($n)
        text $w.txt -state disabled
        pack $w.txt
```

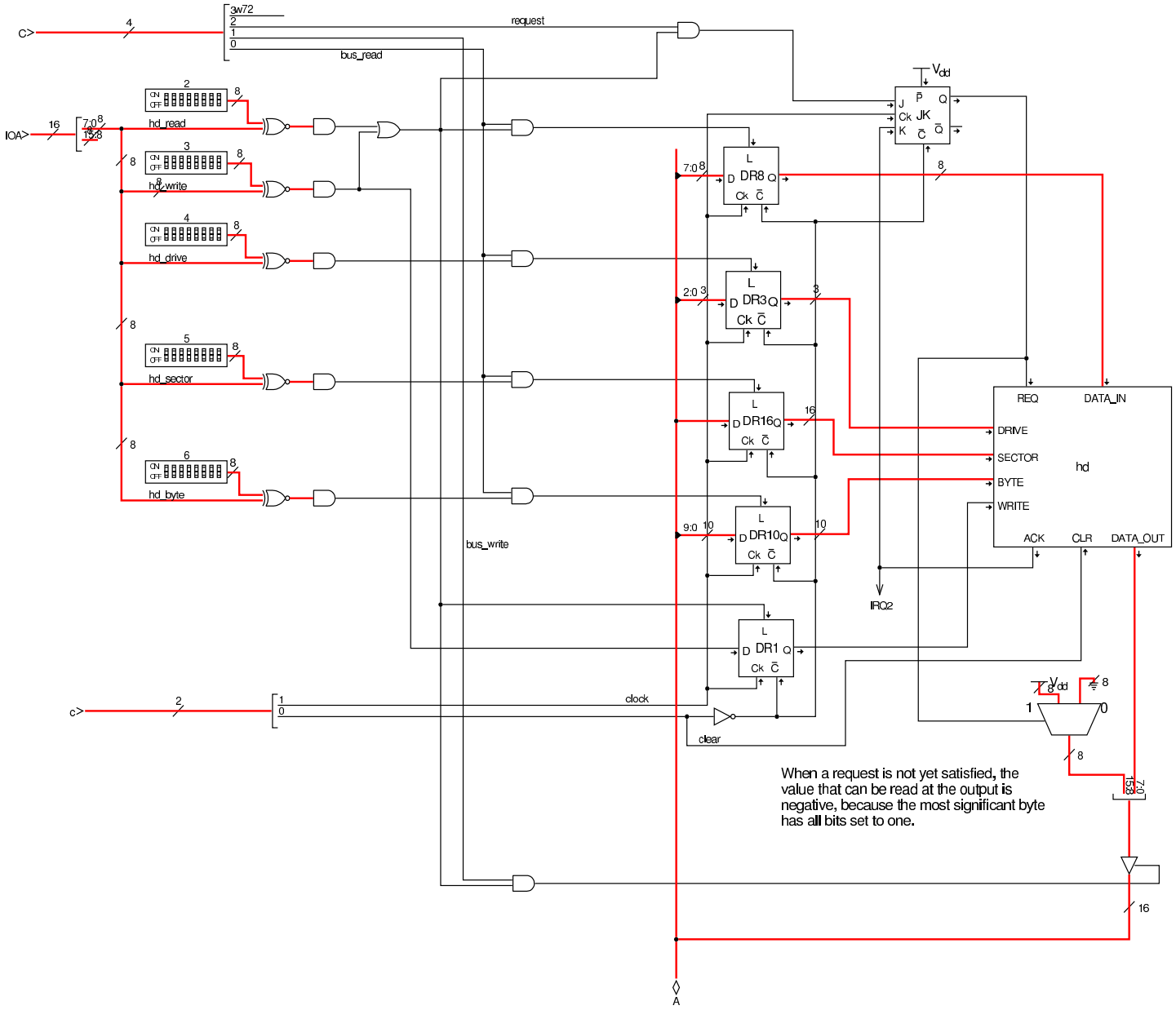
```
        # Put the cursor at the end of the displayed text.
        $w.txt image create end -image txtcurs
        # Bind the keyboard input, related to the object represented by
        # $terminal_w($n), to the function sendChar.
        bind $w <KeyPress> "TERMINAL::sendChar $n \"%A\""
        # Open a reading channel, named «SD» (screen data),
        # and associate it to the function «data»; moreover, open a
        # writing channel, named «KD» (keyboard data).
        if {[info exists ::tkgate_isInitialized]} {
            VPD::outsignal $n.KD TERMINAL::KD($n)
            VPD::insignal $n.SD -command "TERMINAL::data $n" -format %d
        }
        # Reset the character count, used to count the characters displayed
        # on screen.
        set terminal_pos($n) 0
    }
    # Function that receives the typing and put it into the
    # channel «KD», related to the current interface instance.
    proc sendChar {n key} {
        variable KD
        if { [string length $key ] == 1 } {
            binary scan $key c c
            set TERMINAL::KD($n) $c
        }
    }
    # Function that TKGate requires to destroy the interface.
    proc unpost {n} {
        variable terminal_w
        variable terminal_pos
        destroy $terminal_w($n)
        destroy $terminal_pos($n)
        unset terminal_w($n)
        unset terminal_pos($n)
    }
    # Function used to get the data to display on screen.
    proc data {n c} {
        variable terminal_w
        variable terminal_pos
        # For convenience, copy the object reference representing the
        # interface, inside the variable $w.
        set w $terminal_w($n)
        catch {
            # The variable $c contains the character to display.
            if { $c == 7 } {
                # BEL
                bell
                return
            } elseif { $c == 127 || $c == 8 } {
                # DEL | BS
                if { $terminal_pos($n) > 0 } {
                    # Delete the last displayed character, but only if the
                    # characters counter is greater than zero, otherwise
```

```
                    # the cursor would disappear and the next characters
                    # would be located in an unvisible screen area.
                    $w.txt configure -state normal
                    $w.txt delete "end - 3 chars"
                    $w.txt see end
                    $w.txt configure -state disabled
                    set terminal_pos($n) [expr {$terminal_pos($n) - 1}]
                }
                return
            } elseif { $c == 13 } {
                # Convert CR to LF.
                set c 10
            }
            # Convert the character number into a visible symbol.
            set x [format %c $c]
            # Display the symbol.
            $w.txt configure -state normal
            $w.txt insert "end - 2 chars" $x
            $w.txt see end
            $w.txt configure -state disabled
            # Update the displayed characters counter.
            set terminal_pos($n) [expr {$terminal_pos($n) + 1}]
        }
    }
}
```

11.11 Module “HDD”

The **HDD** module is just the same as the previous version: it is a simulated mass-memory drive whith the ability to handle eight units. All the details about the module functionality should be found at the previous version description.

Figure 11.30. Module **HDD**.



Listing 11.31. Verilog code describing the module **hd**.

```
module hd(DRIVE, SECTOR, BYTE, WRITE, DATA_IN, DATA_OUT, REQ, ACK, CLR);
input [2:0] DRIVE;
input WRITE, REQ, CLR;
input [15:0] SECTOR;
input [9:0] BYTE;
input [7:0] DATA_IN;
output [7:0] DATA_OUT;
output ACK;
//
```

```
integer _data_out;
integer _ack;
//
reg [7:0] buffer[0:1023];
reg [8*24-1:0] filename = "hd0_sector_000000000.mem";
//
integer i;
integer sector_8;
integer sector_7;
integer sector_6;
integer sector_5;
integer sector_4;
integer sector_3;
integer sector_2;
integer sector_1;
integer sector_0;
integer x;
//
  initial
    begin
      for (i=0; i<1024; i=i+1)
        begin
          //
          // Initial buffer reset with 00.
          //
          buffer[i] = 8'h00;
        end
      _ack = 0;
      _data_out = 0;
      x = 0;
    end
  //
  always
    begin
      @(posedge CLR)
      _ack = 0;
      _data_out = 0;
      x = 0;
    end
  //
```

```
  //
  //
  always
    begin
      //
      // Start after a positive edge from REQ!.
      //
      @(posedge REQ);
      # 10;
      //
      // Define the sector file name.
      //
      x = SECTOR;
      sector_0 = x%10;
      x = x/10;
      sector_1 = x%10;
      x = x/10;
      sector_2 = x%10;
      x = x/10;
      sector_3 = x%10;
      x = x/10;
      sector_4 = x%10;
      x = x/10;
      sector_5 = x%10;
      x = x/10;
      sector_6 = x%10;
      x = x/10;
      sector_7 = x%10;
      x = x/10;
      sector_8 = x%10;
      //
      // The string starts from the right side and continues
      // to the left!
      //
      filename[12*8+7:12*8] = sector_8 + 8'd48;
      filename[11*8+7:11*8] = sector_7 + 8'd48;
      filename[10*8+7:10*8] = sector_6 + 8'd48;
      filename[9*8+7:9*8] = sector_5 + 8'd48;
      filename[8*8+7:8*8] = sector_4 + 8'd48;
      filename[7*8+7:7*8] = sector_3 + 8'd48;
```

```
        filename[6*8+7:6*8] = sector_2 + 8'd48;
        filename[5*8+7:5*8] = sector_1 + 8'd48;
        filename[4*8+7:4*8] = sector_0 + 8'd48;
        //
        filename[21*8+7:21*8] = DRIVE + 8'd48;
        //
        if (WRITE)
          begin
            //
            // Put data inside the buffer.
            //
            buffer[BYTE] = DATA_IN;
            //
            // Save the buffer to disk.
            // Please remember that $writememh() must be enabled inside
            // TKGate configuration!
            //
            $writememh(filename, buffer);
            //
            // Return the same data read.
            //
            _data_out = buffer[BYTE];
          end
        else
          begin
            //
            // Get data from disk to the buffer.
            //
            $readmemh(filename, buffer);
            //
            // Return the data required.
            //
            _data_out = buffer[BYTE];
          end
        //
        // Acknowledge.
        //
        _ack = 1;
        //
        // Wait the end of request (the negative edge)
```

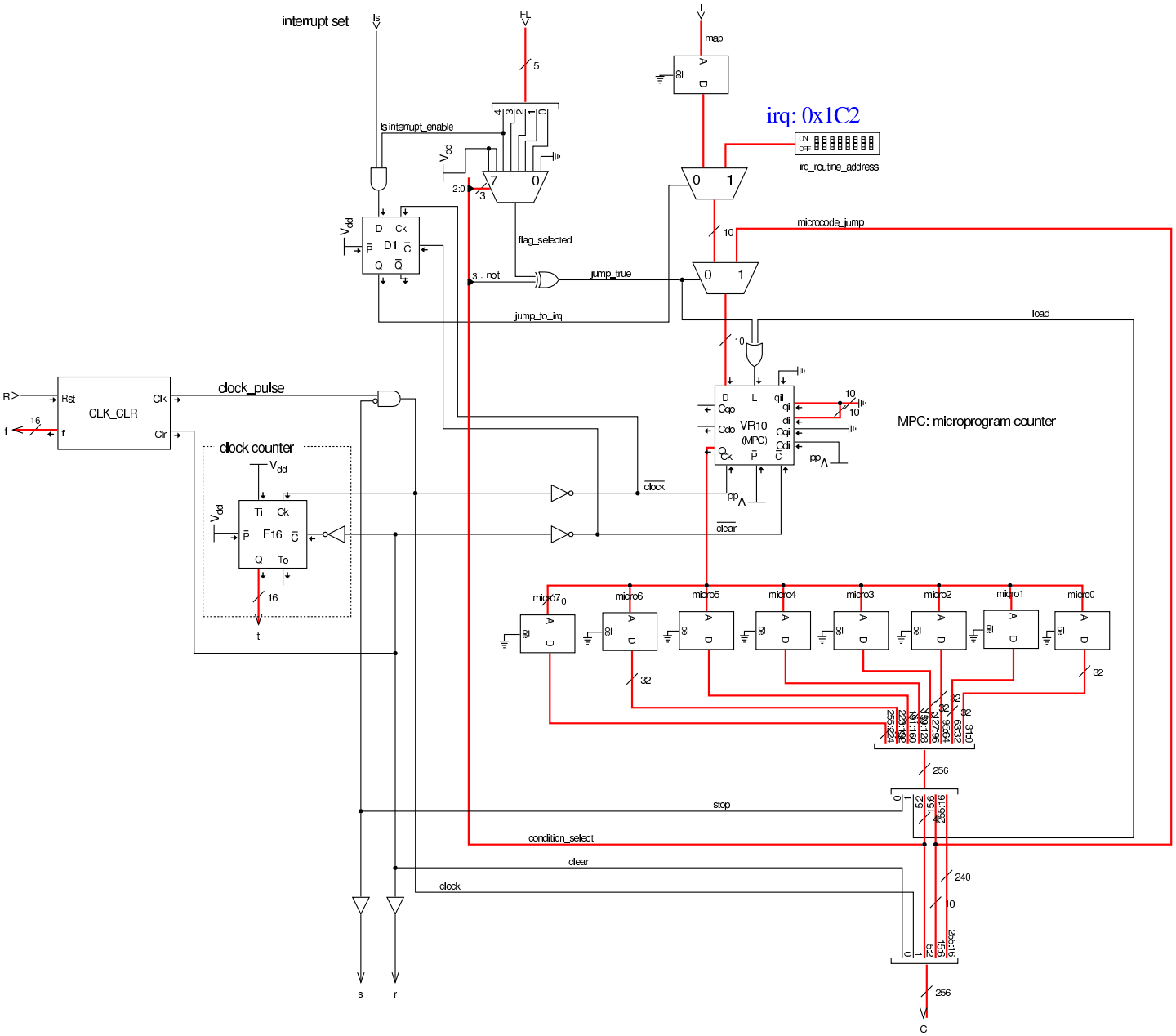
```
      // before restarting the loop.
      //
      @(negedge REQ);
      # 10;
      //
      // Now become ready again.
      //
      _ack = 0;
    end
    //
  assign DATA_OUT = _data_out;
  assign ACK = _ack;
  //
endmodule
```

11.12 Module "CTRL"

«

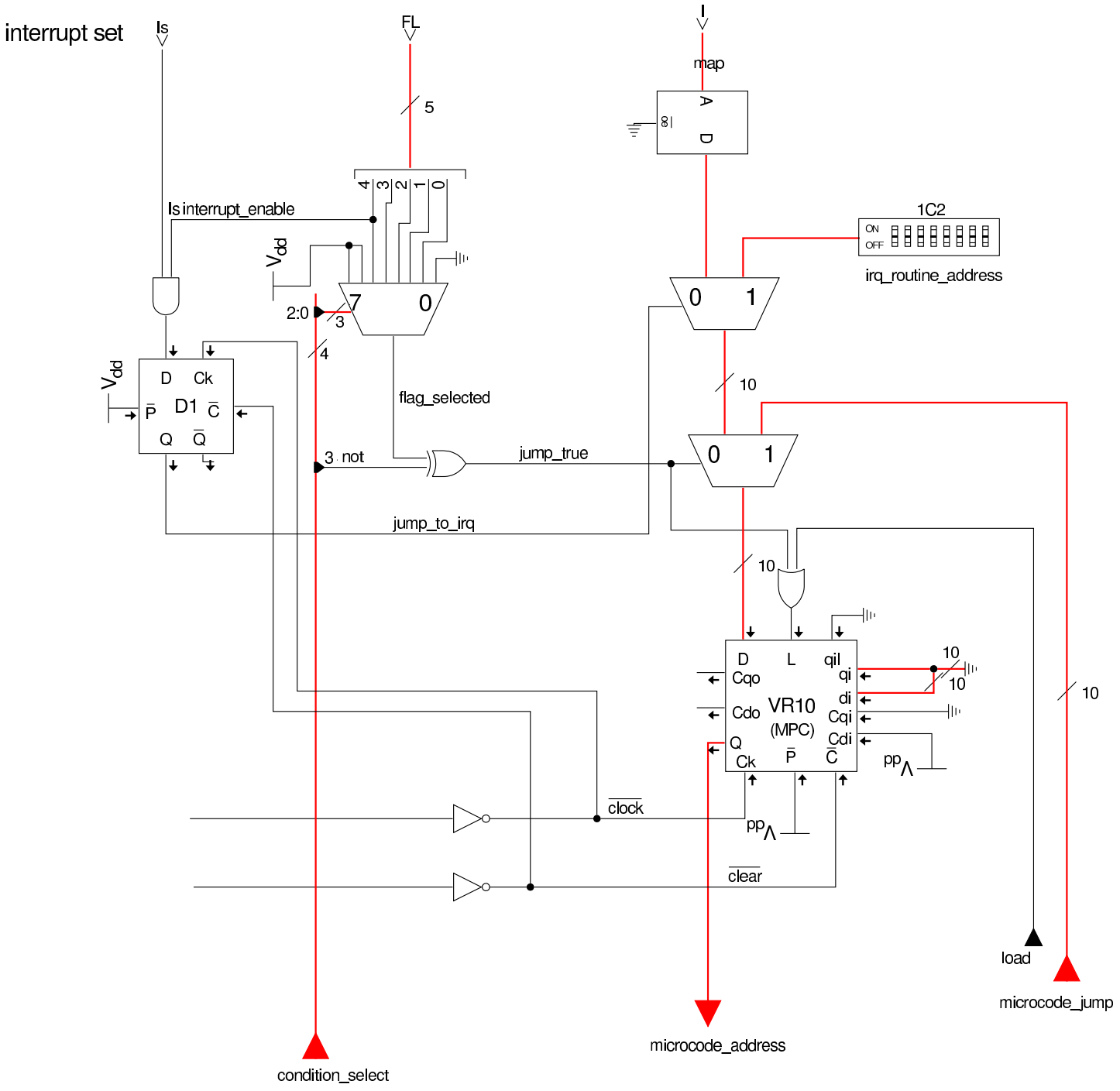
The module **CTRL** is more complex than the previous version, because the conditional jumps are now managed inside the control module and because that requires the ability to jump inside the microcode execution.

Figure 11.32. Module **CTRL**.



The microprogram counter, ***MPC***, is made with a module **VR10**, but connected in the way that it is incremented at every positive clock edge. The ***MPC*** can load a new address when ***load*** line is activated, or when the ***jump_true*** line is active. The ***microcode_jump*** line contains a microcode address to jump to, when ***jump_true*** is active. The ***condition_select*** line contains a number, representing the condition to be checked, so that, if true, will activate the line ***jump_true***.

Figure 11.33. Module **CTRL**: microcode load selection.



The previous figure shows the detail of the upper side, where there is the selection control of the microcode address to load inside the ***MPC***. The microcode address might come:

- from the map memory, as a translation of the opcode into microcode address;
- from the dip-switch containing the address of the microcode procedure necessary

to handle a hardware interrupt (IRQ);

- from the line ***microcode_jump***.

When no other special conditions are present and the line ***load*** is active, the register ***MPC*** loads the value obtained from the map memory. If a hardware interrupt was previously received while hardware interrupts were allowed, it is stored inside the D flip-flop on the right. Then, when the control unit tries to load the next opcode address, it loads the interrupt procedure address instead. The ***condition_select*** and the ***microcode_jump*** lines are controlled by the microcode: this way the jump that is selected is done only if the condition results true, otherwise, there is no jump and the register ***MPC*** is simply incremented to the next microinstruction.

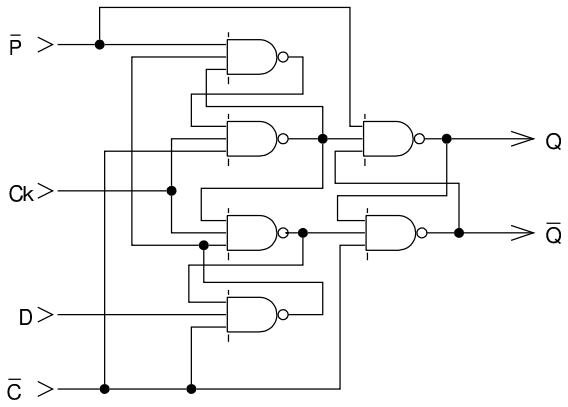
The control unit contains also the clock generator and a frequency divisor, as described in the following sections.

11.12.1 Modules "F*n*"

The modules **`F`*n*** are frequency divisors, made with T flip-flop, as the following figures show.

Figure 11.34. Modules **`D1`** and **`T1`**: building a T flip-flop from a D flip-flop.

module D1: positive edge D flip–flop



module T1: T flip–flop

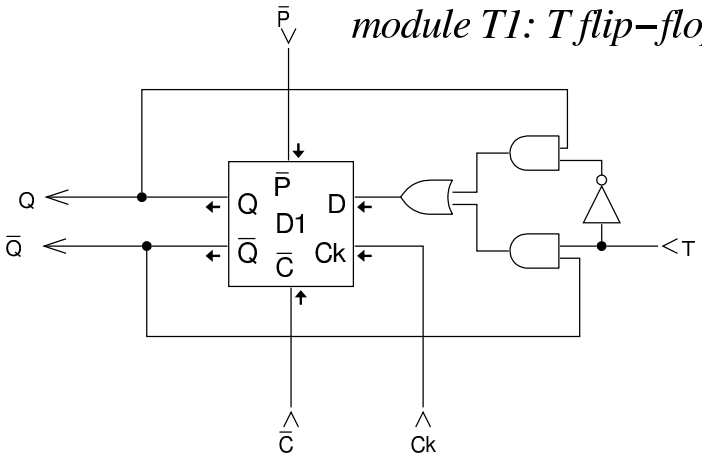
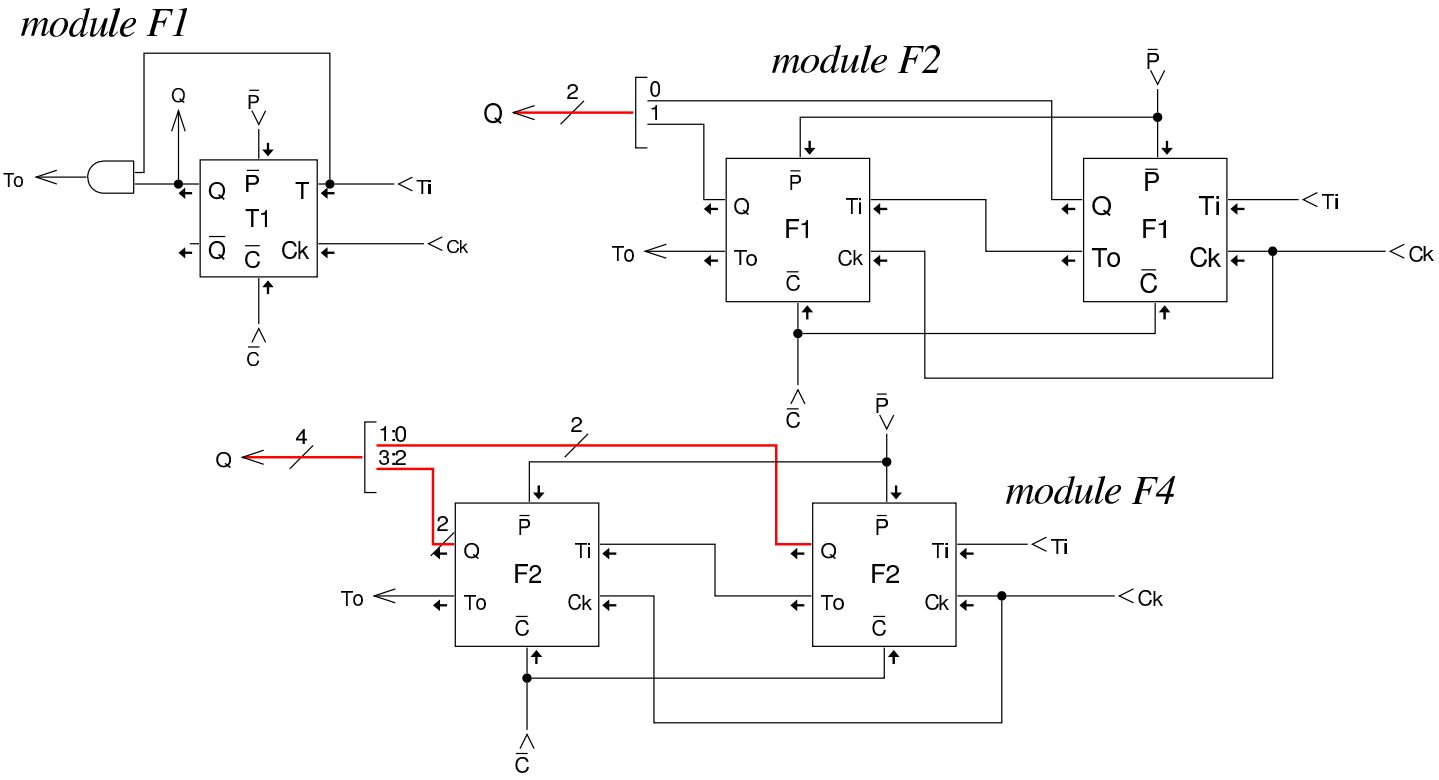


Figure 11.35. Building the **Fn** modules.



11.12.2 Module "CLK_CLR"

The module **CLK_CLR** is responsible for the clock pulse generation and for the reset line, which is to be synchronized with the clock. The clock frequency may be controlled by the dip-switch at the top. It is important to remind that the control unit receives an inverted clock pulse, because the microcode data should be ready before the clock positive edge.

Figure 11.36. Module **CLK_CLR**.

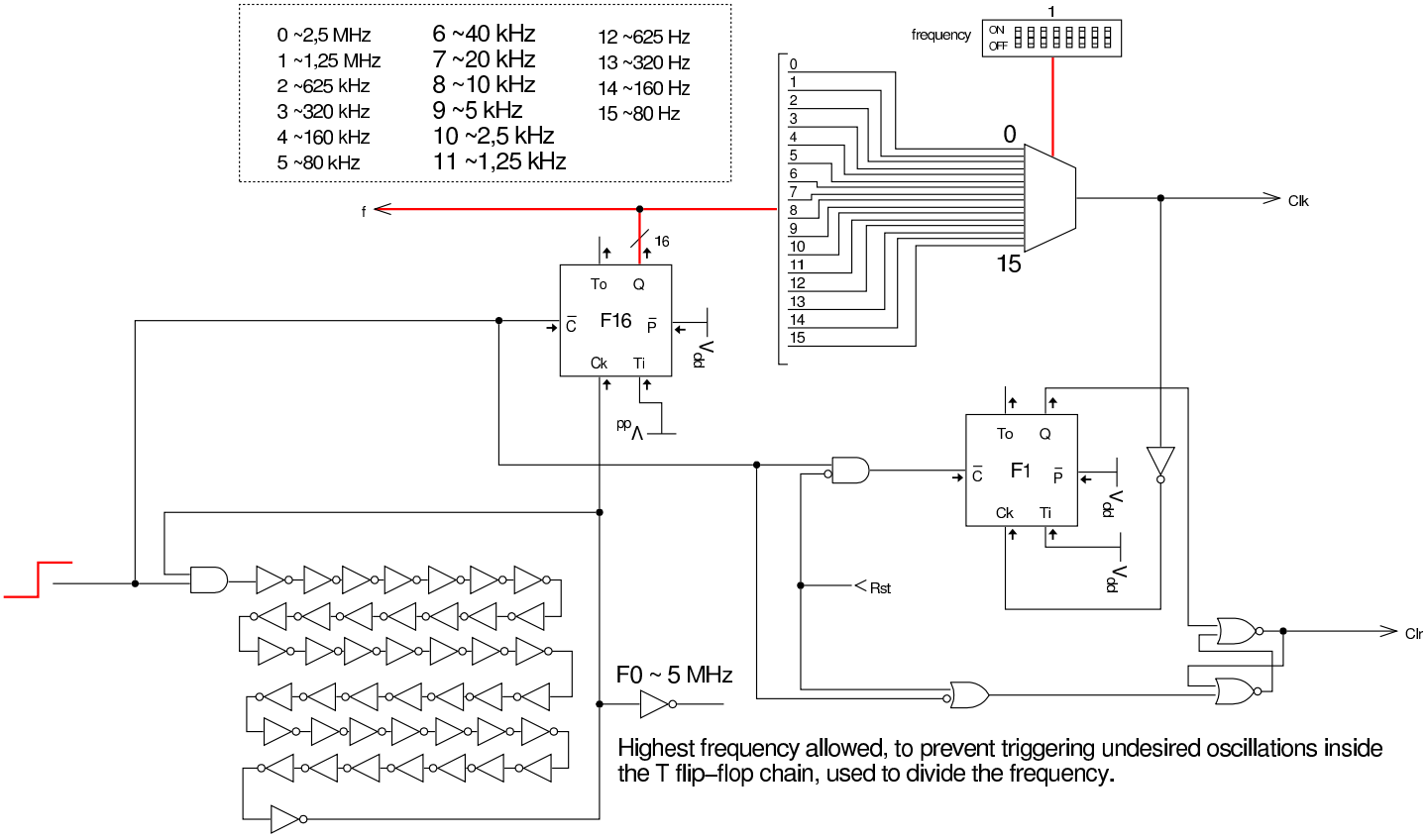


Figure 11.37. Module **one_up**, used to start the oscillation inside the module **CLK_CLR**.

```
module one_up #(.W(1000)) (Z);
  output Z;
  reg Z;

  initial
    begin
      Z = 1'b0;
      $tkg$wait(W);
      Z = 1'b1;
    end

endmodule
```

11.13 Memory and microcode fields

«

The memory for this version of the project are described by the following lines of TKGate code. There are many blocks of memory for the microcode word, because every component connected to the data-bus can be controlled independently.

Listing 11.38. Memory banks description for TKGate.

```
map       bank[9:0]     ctrl.map;
microcode bank[31:0]    ctrl.micro0;
microcode bank[63:32]   ctrl.micro1;
microcode bank[95:64]   ctrl.micro2;
microcode bank[127:96]  ctrl.micro3;
microcode bank[159:128] ctrl.micro4;
microcode bank[191:160] ctrl.micro5;
microcode bank[223:192] ctrl.micro6;
microcode bank[255:224] ctrl.micro7;
macrocode bank[15:0]    ram.ram;
```

Listing 11.39. Fields of the microcode word, for TKGate.

```
field ctrl[1:0]    = {nop=0, stop=1, load=2};
field jump[5:2]    = {false=0, carry_t=1, zero_t=2, negative_t=3,
                      overflow_t=4, irq_enabled_t=5, true=6,
                      carry_f=9, zero_f=10, negative_f=11,
                      overflow_f=12, irq_enabled_f=13 };
field addr[15:6]   = {ciao=0};
//
field ir[31:16]    = {select_bus_a=0, select_bus_b=1, bus_write=2,
                      register_load_byte_1=4, register_load_byte_2=8,
                      register_load=12,
                      select_byte_1=0, select_byte_2=16,
                      rank_8=0, rank_16=32, unsigned=0, signed=64,
                      qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                      qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                      qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                      qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                      dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                      dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                      dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                      dm3=0xD000, dm2=0xE000, dm1=0xF000 };
```

```
//
field ram[37:32]    = {select_bus_a=0, select_bus_b=1, bus_write=2,
                       load=4, p=0, i=8, j=16, s=24, t=32};
//
field pc[53:38]     = {select_bus_a=0, select_bus_b=1, bus_write=2,
                       register_load_byte_1=4, register_load_byte_2=8,
                       register_load=12,
                       select_byte_1=0, select_byte_2=16,
                       rank_8=0, rank_16=32, unsigned=0, signed=64,
                       qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                       qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                       qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                       qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                       dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                       dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                       dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                       dm3=0xD000, dm2=0xE000, dm1=0xF000 };
field i[69:54]      = {select_bus_a=0, select_bus_b=1, bus_write=2,
                       register_load_byte_1=4, register_load_byte_2=8,
                       register_load=12,
                       select_byte_1=0, select_byte_2=16,
                       rank_8=0, rank_16=32, unsigned=0, signed=64,
                       qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                       qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                       qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                       qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                       dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                       dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                       dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                       dm3=0xD000, dm2=0xE000, dm1=0xF000 };
field j[85:70]      = {select_bus_a=0, select_bus_b=1, bus_write=2,
                       register_load_byte_1=4, register_load_byte_2=8,
                       register_load=12,
                       select_byte_1=0, select_byte_2=16,
                       rank_8=0, rank_16=32, unsigned=0, signed=64,
                       qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                       qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                       qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                       qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                       dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
```

```
                        dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                        dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                        dm3=0xD000, dm2=0xE000, dm1=0xF000 };
field sp[101:86]   = {select_bus_a=0, select_bus_b=1, bus_write=2,
                        register_load_byte_1=4, register_load_byte_2=8,
                        register_load=12,
                        select_byte_1=0, select_byte_2=16,
                        rank_8=0, rank_16=32, unsigned=0, signed=64,
                        qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                        qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                        qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                        qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                        dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                        dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                        dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                        dm3=0xD000, dm2=0xE000, dm1=0xF000 };
field tmp[117:102] = {select_bus_a=0, select_bus_b=1, bus_write=2,
                        register_load_byte_1=4, register_load_byte_2=8,
                        register_load=12,
                        select_byte_1=0, select_byte_2=16,
                        rank_8=0, rank_16=32, unsigned=0, signed=64,
                        qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                        qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                        qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                        qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                        dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                        dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                        dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                        dm3=0xD000, dm2=0xE000, dm1=0xF000 };
field fl[133:118]  = {select_bus_a=0, select_bus_b=1, bus_write=2,
                        register_load_byte_1=4, register_load_byte_2=8,
                        register_load=12,
                        select_byte_1=0, select_byte_2=16,
                        rank_8=0, rank_16=32, unsigned=0, signed=64,
                        qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                        qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                        qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                        qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                        dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                        dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
```

```
                        dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                        dm3=0xD000, dm2=0xE000, dm1=0xF000 };
field c[149:134]    = {select_bus_a=0, select_bus_b=1, bus_write=2,
                        register_load_byte_1=4, register_load_byte_2=8,
                        register_load=12,
                        select_byte_1=0, select_byte_2=16,
                        rank_8=0, rank_16=32, unsigned=0, signed=64,
                        qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                        qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                        qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                        qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                        dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                        dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                        dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                        dm3=0xD000, dm2=0xE000, dm1=0xF000 };
//
field alu[158:150] = {unsigned=0, signed=1, rank_8=0, rank_16=2,
                        a=0, and=4, or=8, xor=12,
                        nxor=16, nor=20, nand=24, not=28,
                        add=64, sub=68, addc=72, subb=76,
                        lshl=128, lshr=132, ashl=136, ashr=140,
                        rotcl=144, rotcr=148,
                        rotl=192, rotr=196,
                        clearc=256, clearz=260, clearn=264,
                        clearo=268, cleari=272,
                        setc=288, setz=292, setn=296,
                        seto=300, seti=304};
//
field a[174:159]    = {select_bus_a=0, select_bus_b=1, bus_write=2,
                        register_load_byte_1=4, register_load_byte_2=8,
                        register_load=12,
                        select_byte_1=0, select_byte_2=16,
                        rank_8=0, rank_16=32, unsigned=0, signed=64,
                        qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                        qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                        qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                        qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                        dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                        dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                        dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
```

```
                        dm3=0xD000, dm2=0xE000, dm1=0xF000 };
field b[190:175]    = {select_bus_a=0, select_bus_b=1, bus_write=2,
                        register_load_byte_1=4, register_load_byte_2=8,
                        register_load=12,
                        select_byte_1=0, select_byte_2=16,
                        rank_8=0, rank_16=32, unsigned=0, signed=64,
                        qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                        qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                        qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                        qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                        dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                        dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                        dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                        dm3=0xD000, dm2=0xE000, dm1=0xF000 };
field bp[206:191]   = {select_bus_a=0, select_bus_b=1, bus_write=2,
                        register_load_byte_1=4, register_load_byte_2=8,
                        register_load=12,
                        select_byte_1=0, select_byte_2=16,
                        rank_8=0, rank_16=32, unsigned=0, signed=64,
                        qp1=0x0080, qp2=0x0100, qp3=0x0180, qp4=0x0200,
                        qp5=0x0280, qp6=0x0300, qp7=0x0380, qm8=0x0400,
                        qm7=0x0480, qm6=0x0500, qm5=0x0580, qm4=0x0600,
                        qm3=0x0680, qm2=0x0700, qm1=0x0780, qup=0x0800,
                        dp1=0x1000, dp2=0x2000, dp3=0x3000, dp4=0x4000,
                        dp5=0x5000, dp6=0x6000, dp7=0x7000, dm8=0x8000,
                        dm7=0x9000, dm6=0xA000, dm5=0xB000, dm4=0xC000,
                        dm3=0xD000, dm2=0xE000, dm1=0xF000 };
//
field bus[224:207] = {bw=0x10000, select_bus_a=0, select_bus_b=0x20000};
//
field ioa[240:225] = {select_bus_a=0, select_bus_b=1, bus_write=2,
                        register_load_byte_1=4, register_load_byte_2=8,
                        register_load=12,
                        select_byte_1=0, select_byte_2=16,
                        rank_8=0, rank_16=32, unsigned=0, signed=64,
                        p1=128, p2=256, p3=384, p4=512,
                        p5=640, p6=768, p7=896,
                        m8=1024, m7=1152, m6=1280, m5=1480, m4=1536,
                        m3=1664, m2=1792, m1=1920, update=2048};
//
```

```
field ivt[243:241] = {bus_write=1, ivt_load=2,
                      select_int_a=0, select_int_b=4};
//
field irq[246:244] = {bus_write=1, irq_mask_load=2, irq_done=4};
//
field ioc[250:247] = {load=1, bus_write=2, req=4, isack=8};
```

11.14 Opcodes

For the opcodes are used only eight bits and all available combinations are specified. The following table shows the opcode organization.

Table 11.40. Opcodes.

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0x00	00......	cp % %		%			%					
0x40	010000..	cp n %						A, B, I, J		const16		
0x44	010001..	cp (%) %						(I, J)	A, B			
0x48	010010..	cp % (%)						A, B	(I, J)			
0x4C	0100110.	cp n (%)							(I, J)	const16		
0x4E	0100111.	cp (%) (%)							(I, J)			
0x50	010100..	cp8 n %						A, B, I, J		const8		
0x54	010101..	cp8 (%) %						(I, J)	A, B			
0x58	010110..	cp8 % (%)						A, B	(I, J)			
0x5C	0101110.	cp8 n (%)							(I, J)	const8		
0x5E	0101111.	cp8 (%) (%)							(I, J)			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0x60	01100...	push %					%					
0x68	01101...	push8 %					%					
0x70	01110...	pop %					%					
0x78	01111...	pop8 %					%					
0x80	10000000	push n								const16		
0x81	10000001	push8 n								const8		
0x82	1000001.	push (%)							(I, J)			
0x84	1000010.	push8 (%)							(I, J)			
0x86	1000011.	pop (%)							(I, J)			
0x88	1000100.	pop8 (%)							(I, J)			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0x90	10010...	jump %flag 0|1 #ref					%flag		TRUE	#ref		
0x98	1001100.	equal, not							NOT			
0x9A	1001101.	and							NOT			
0x9C	1001110.	or							NOT			
0x9E	1001111.	xor							NOT			
0xA0	1010000.	add, sub							sub			
0xA2	1010001.	addc, subb							sub			
0xA4	1010010.	lsh							right			
0xA6	1010011.	ash							right			
0xA8	1010100.	rotc							right			
0xAA	1010101.	rot							right			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0xAC	1010110.	add8, sub8							sub			
0xAE	1010111.	add8c, sub8b							sub			
0xB0	1011000.	lsh8							right			
0xB2	1011001.	ash8							right			
0xB4	1011010.	rot8c							right			
0xB6	1011011.	rot8							right			
0xB8	101110..	cast						sign	A, B			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0xBC	1011110.	equals, nots							NOT			
0xBE	1011111.	ands							NOT			
0xC0	1100000.	ors							NOT			
0xC2	1100001.	xors							NOT			
0xC4	1100010.	adds, subs							sub			
0xC6	1100011.	addcs, subbs							sub			
0xC8	1100100.	lshls, lshrs							right			
0xCA	1100101.	ashls, ashrs							right			
0xCC	1100110.	rotcls, rotcrs							right			
0xCE	1100111.	rotls, rotrs							right			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0xD0	1101000.	add8s, sub8s							sub			
0xD2	1101001.	add8cs, sub8bs							sub			
0xD4	1101010.	lsh8ls, lsh8rs							right			
0xD6	1101011.	ash8ls, ash8rs							right			
0xD8	1101100.	rot8cls, rot8crs							right			
0xDA	1101101.	rot8ls, rot8rs							right			
0xDC	1101110.	casts							sign			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0xDE	11011110	call								#ref		
0xDF	11011111	return										
0xE0	11100000	int								int		
0xE1	11100001	iret										
0xE2	11100010	imrl								mask		
0xE3	11100011	ivtl								#ref		
0xE4	1110010.	flag_i 0|1							set			
0xE6	1110011.	flag_c 0|1							set			
0xE8	1110100.	in n %							A, B	port		
0xEA	1110101.	in n (%)							(I, J)	port		
0xEC	1110110.	out % n							A, B	port		
0xEE	1110111.	out (%) n							(I, J)	port		
0xF0	11110000	if_ack_jump								port	#ref	
0xF1	11110001	if_ack_call								port	#ref	

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0xF2	1111001.	cmpr, testr							test			
0xF4	1111010.	cmpi, testi							test			
0xF6	1111011.	cmps, tests							test			
0xF8	1111100.	cmp8i, test8i							test			
0xFA	1111101.	cmp8s, test8s							test			
0xFC	1111110.	call (%)							I, J			
0xFE	11111110	jump								#ref		
0xFF	11111111	stop										

Table 11.41. Macrocode syntax.

Macrocode syntax	Description
`nop`	Not operate.
`cp %`***src***`, %`***dst***	Copy the ***src*** register content inside the ***dst*** register. Allowed registers are: ***I***, ***J***, ***A***, ***B***, ***BP***, ***SP***, ***C***, ***FL***.
`cp` ***n16***`, %`***dst***	Assign a 16-bit constant to a register. Allowed destination registers are: ***I***, ***J***, ***A***, ***B***.
`cp (%I|%J), %A|%B`	Assign to register ***A*** or ***B*** the 16-bit value contained in memory at the address already available from register ***I*** or ***J***.
`cp %A|%B, (%I|%J)`	Copy in memory the 16-bit value of register ***A*** or ***B***, to the location represented by the register ***I*** or ***J***.
`cp` ***n16***`, (%I|%J)`	Assign in memory the 16-bit constant, to the location represented by the register ***I*** or ***J***.

Macrocode syntax	Description
`cp (%I)`	Copy the 16-bit memory content at the address represented by register ***I***, to the memory location represented by the register ***J***.
`cp (%J)`	Copy the 16-bit memory content at the address represented by register ***J***, to the memory location represented by the register ***I***.
`cp8` ***n8***`, %`***dst***	Assign a 8-bit constant to a register. Allowed destination registers are: ***I***, ***J***, ***A***, ***B***.
`cp8 (%I|%J), %A|%B`	Assign to register ***A*** or ***B*** the 8-bit value contained in memory at the address already available from register ***I*** or ***J***.
`cp8 %A|%B, (%I|%J)`	Copy in memory the 8-bit value of register ***A*** or ***B***, to the location represented by the register ***I*** or ***J***.
`cp8` ***n8***`, (%I|%J)`	Assign in memory the 8-bit constant, to the location represented by the register ***I*** or ***J***.
`cp8 (%I)`	Copy the 8-bit memory content at the address represented by register ***I***, to the memory location represented by the register ***J***.
`cp8 (%J)`	Copy the 8-bit memory content at the address represented by register ***J***, to the memory location represented by the register ***I***.
`push` ***reg***	Push at the top of the stack the 16-bit value contained inside the register.
`pop` ***reg***	Pop from the stack a 16-bit value that is then assigned to the register.
`push8` ***reg***	Push at the top of the stack the 8-bit value contained inside the register.
`pop8` ***reg***	Pop from the stack a 8-bit value that is then assigned to the register.

Macrocode syntax	Description
`push` ***n16***	Push at the top of the stack the 16-bit constant.
`push` ***n8***	Push at the top of the stack the 8-bit constant.
`push (%I|%J)`	Push at the top of the stack the 16-bit value contained in memory, at the location represented by register ***I*** or ***J***.
`push (%I|%J)`	Push at the top of the stack the 16-bit value contained in memory, at the location represented by register ***I*** or ***J***.
`push8 (%I|%J)`	Push at the top of the stack the 8-bit value contained in memory, at the location represented by register ***I*** or ***J***.
`pop (%I|%J)`	Pop from the top of the stack a 16-bit that is then copied at the memory location represented by register ***I*** or ***J***.
`pop8 (%I|%J)`	Pop from the top of the stack a 8-bit that is then copied at the memory location represented by register ***I*** or ***J***.
`jump #`***ref***	Jump to the address specified.

`jump %carry|%borrow↩`
`↪ |%zero|%negative↩`
`↪ |%overflow, ↩`
`↪ 0|1, #`***ref***
	Conditional jump. The first argument is a pseudo-register that selects the flag; the second argument means the value that the flag should have; the third argument is the address where the jump should go if the condition is met.

`equal`	***C*** = ***A***, updating the flags status
`not`	***C*** = ~***A***, updating the flags status (one's complement).
`and`	***C*** = ***A*** & ***B***, updating the flags.
`nand`	***C*** = ~(***A*** & ***B***), updating the flags.

Macrocode syntax	Description
`or`	***C*** = ***A*** | ***B***, updating the flags.
`nor`	***C*** = ~(***A*** | ***B***), updating the flags.
`xor`	***C*** = ***A*** ^ ***B***, updating the flags.
`nxor`	***C*** = ~(***A*** ^ ***B***), updating the flags.
`add`	***C*** = ***A*** + ***B***, updating the flags, 16-bit.
`sub`	***C*** = ***A*** − ***B***, updating the flags, 16-bit.
`addc`	***C*** = ***A*** + ***B*** + carry, updating the flags, 16-bit.
`subb`	***C*** = ***A*** − ***B*** − borrow, updating the flags, 16-bit.
`lshl`	***C*** = lshl(***A***), updating the flags: logic shift left, 16-bit.
`lshr`	***C*** = lshr(***A***), updating the flags: logic shift right, 16-bit.
`ashl`	***C*** = ashl(***A***), updating the flags: arithmetic shift left, 16-bit.
`ashr`	***C*** = ashr(***A***), updating the flags: arithmetic shift right, 16-bit.
`rotcl`	***C*** = rotcl(***A***), updating the flags: rotation left whith carry, 16-bit.
`rotcr`	***C*** = rotcr(***A***), updating the flags: rotation right whith carry, 16-bit.
`rotl`	***C*** = rotl(***A***), updating the flags: rotation left, 16-bit.
`rotr`	***C*** = rotr(***A***), updating the flags: rotation right, 16-bit.

Macrocode syntax	Description
`add8`	***C*** = ***A*** + ***B***, updating the flags, 8-bit.
`sub8`	***C*** = ***A*** − ***B***, updating the flags, 8-bit.
`add8c`	***C*** = ***A*** + ***B*** + carry, updating the flags, 8-bit.
`sub8b`	***C*** = ***A*** − ***B*** − borrow, updating the flags, 8-bit.
`lsh8l`	***C*** = lsh8l(***A***), updating the flags: logic shift left, 8-bit.
`lsh8r`	***C*** = lsh8r(***A***), updating the flags: logic shift right, 8-bit.
`ash8l`	***C*** = ash8l(***A***), updating the flags: arithmetic shift left, 8-bit.
`ash8r`	***C*** = ash8r(***A***), updating the flags: arithmetic shift right, 8-bit.
`rot8cl`	***C*** = rot8cl(***A***), updating the flags: rotation left whith carry, 8-bit.
`rot8cr`	***C*** = rot8cr(***A***), updating the flags: rotation right whith carry, 8-bit.
`rot8l`	***C*** = rot8l(***A***), updating the flags: rotation left, 8-bit.
`rot8r`	***C*** = rot8r(***A***), updating the flags: rotation right, 8-bit.
`cast 0|1 %A|%B`	Adapt the register value to 8-bit, unsigned or signed, depending on the first argument.
`equals`	Read the 16-bit value on top of the stack and update the flags.
`nots`	Read the 16-bit value on top of the stack and replace it with the one's complement, updating the flags.

Macrocode syntax	Description
`ands`	Pop two 16-bit values from the top of the stack and push the bitwise AND of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`nands`	Pop two 16-bit values from the top of the stack and push the bitwise NAND of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`ors`	Pop two 16-bit values from the top of the stack and push the bitwise OR of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`nors`	Pop two 16-bit values from the top of the stack and push the bitwise NOR of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`xors`	Pop two 16-bit values from the top of the stack and push the bitwise XOR of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`nxors`	Pop two 16-bit values from the top of the stack and push the bitwise NXOR of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`adds`	Pop two 16-bit values from the top of the stack and push the sum of them, updating the flags. To do the calculation, the registers ***A*** and ***B*** are overwritten.
`subs`	Pop a 16-bit values from the top of the stack and put it inside the register ***B***; then pop another 16-bit value and put inside the register ***A***; then calculate ***A*−*B*** updating the flags and push the result inside the stack again.
`addcs`	Pop two 16-bit values from the top of the stack and push the sum with carry of them, updating the flags. To do the calculation, the registers ***A*** and ***B*** are overwritten.

Macrocode syntax	Description
`subbs`	Pop a 16-bit values from the top of the stack and put it inside the register ***B***; then pop another 16-bit value and put inside the register ***A***; then calculate ***A*−*B***−borrow updating the flags and push the result inside the stack again.
`lshls`	Pop a 16-bit values from the top of the stack and push the logic shift left of it, updating the flags.
`lshrs`	Pop a 16-bit values from the top of the stack and push the logic shift right of it, updating the flags.
`ashls`	Pop a 16-bit values from the top of the stack and push the arithmetic shift left of it, updating the flags.
`ashrs`	Pop a 16-bit values from the top of the stack and push the arithmetic shift right of it, updating the flags.
`rotcls`	Pop a 16-bit values from the top of the stack and push the rotation left with carry of it, updating the flags.
`rotcrs`	Pop a 16-bit values from the top of the stack and push the rotation right with carry of it, updating the flags.
`rotls`	Pop a 16-bit values from the top of the stack and push the rotation left of it, updating the flags.
`rotrs`	Pop a 16-bit values from the top of the stack and push the rotation right of it, updating the flags.
`add8s`	Pop two 8-bit values from the top of the stack and push the sum of them, updating the flags. To do the calculation, the registers ***A*** and ***B*** are overwritten.

Macrocode syntax	Description
`sub8s`	Pop a 8-bit values from the top of the stack and put it inside the register ***B***; then pop another 16-bit value and put inside the register ***A***; then calculate ***A*−*B*** updating the flags and push the result inside the stack again.
`add8cs`	Pop two 8-bit values from the top of the stack and push the sum with carry of them, updating the flags. To do the calculation, the registers ***A*** and ***B*** are overwritten.
`sub8bs`	Pop a 8-bit values from the top of the stack and put it inside the register ***B***; then pop another 16-bit value and put inside the register ***A***; then calculate ***A*−*B***−borrow updating the flags and push the result inside the stack again.
`lsh8ls`	Pop a 8-bit values from the top of the stack and push the logic shift left of it, updating the flags.
`lsh8rs`	Pop a 8-bit values from the top of the stack and push the logic shift right of it, updating the flags.
`ash8ls`	Pop a 8-bit values from the top of the stack and push the arithmetic shift left of it, updating the flags.
`ash8rs`	Pop a 8-bit values from the top of the stack and push the arithmetic shift right of it, updating the flags.
`rotc8ls`	Pop a 8-bit values from the top of the stack and push the rotation left with carry of it, updating the flags.
`rotc8rs`	Pop a 8-bit values from the top of the stack and push the rotation right with carry of it, updating the flags.
`rot8ls`	Pop a 8-bit values from the top of the stack and push the rotation left of it, updating the flags.

Macrocode syntax	Description
`rot8rs`	Pop a 8-bit values from the top of the stack and push the rotation right of it, updating the flags.
`casts 0|1`	Pop a 16-bit value from the top of the stack and push the same value, adapted to 8-bit, unsigned or signed, depending on the argument.
`call #`***ref***	Call the procedure that starts at the argument reference: the registers ***FL*** and ***PC*** are pushed inside the stack.
`call (%I|%J)`	Call the procedure that starts at the position contained inside register ***I*** or ***J***: the registers ***FL*** and ***PC*** are pushed inside the stack.
`return`	Return from a previous call: the registers ***PC*** and ***FL*** are retrieved from the stack.
`int` ***n8***	Software interrupt call: the registers ***FL*** and ***PC*** are pushed inside the stack.
`iret`	Return from an interrupt: the registers ***PC*** and ***FL*** are retrieved from the stack.
`imrl` ***n8***	Load the IMR: interrupt mask register.
`ivtl #`***ref***	Load the IVT: interrupt vector table. The argument is the interrupt vector table start address in memory.
`flag_i 0|1`	Reset or set the flag that allow the hardware interrupts.
`flag_c 0|1`	Reset or set the carry (borrow) flag.
`in` ***n8*****`,` `%A|%B`	Read from the I/O address specified, a byte that is copied inside register ***A*** or ***B***.
`in` ***n8*****`,` `(%I|%J)`	Read from the I/O address specified, a byte that is copied in memory, at the address specified by register ***I*** or ***J***.

Macrocode syntax	Description
`out %A|%B,` ***n8***	Write to the I/O address specified, a byte that is read from register ***A*** or ***B***.
`out (%I|%J),` ***n8***	Write to the I/O address specified, a byte that is read from memory, at the address specified by register ***I*** or ***J***.
`ifack_jump` ***n8*****,** `#`***ref***	If the I/O port gives an acknoledge report, then jump to the specified address.
`ifack_call` ***n8*****,** `#`***ref***	If the I/O port gives an acknoledge report, then call the specified procedure.
`cmpr`	Calculate ***A−B***, updating the flags, without modifying the accumulator.
`testr`	Calculate ***A&B*** (bitwise AND), updating the flags, without modifying the accumulator.
`cmpi`	Load ***A*** with the value contained in memory at the address represented by the register ***I***; load ***B*** with the value contained in memory at the address represented by the register ***J***; then calculate ***A−B***, updating the flags, without modifying the accumulator.
`testi`	Load ***A*** with the value contained in memory at the address represented by the register ***I***; load ***B*** with the value contained in memory at the address represented by the register ***J***; then calculate ***A&B*** (bitwise AND), updating the flags, without modifying the accumulator.
`cmps`	Read from the top of the stack a 16-bit value and put it inside the register ***B***; read from the stack, just under the previous value, a 16-bit value and put it inside the register ***A***; then calculate ***A−B***, updating the flags, without modifying the accumulator.

Macrocode syntax	Description
`tests`	Read from the top of the stack a 16-bit value and put it inside the register ***B***; read from the stack, just under the previous value, a 16-bit value and put it inside the register ***A***; then calculate ***A&B***, updating the flags, without modifying the accumulator.
`cmp8r`	Calculate 8-bit ***A−B***, updating the flags, without modifying the accumulator.
`test8r`	Calculate 8-bit ***A&B*** (bitwise AND), updating the flags, without modifying the accumulator.
`cmp8i`	Load 8-bit inside ***A*** with the value contained in memory at the address represented by the register ***I***; load 8-bit inside ***B*** with the value contained in memory at the address represented by the register ***J***; then calculate ***A−B***, updating the flags, without modifying the accumulator.
`test8i`	Load 8-bit inside ***A*** with the value contained in memory at the address represented by the register ***I***; load 8-bit inside ***B*** with the value contained in memory at the address represented by the register ***J***; then calculate ***A&B***, updating the flags, without modifying the accumulator.
`cmp8s`	Read from the top of the stack a 8-bit value and put it inside the register ***B***; read from the stack, just under the previous value, a 8-bit value and put it inside the register ***A***; then calculate ***A−B***, updating the flags, without modifying the accumulator.

Macrocode syntax	Description
`test8s`	Read from the top of the stack a 8-bit value and put it inside the register ***B***; read from the stack, just under the previous value, a 8-bit value and put it inside the register ***A***; then calculate ***A*** &***B***, updating the flags, without modifying the accumulator.
`stop`	Stop the system.

Listing 11.42. Registers, flags and opcodes encoded for TKGate.

```
registers I=0, J=1, A=2, B=3, BP=4, SP=5, C=6, FL=7;

registers carry=0, borrow=0, zero=1, negative=2, overflow=3;
//-----------------------------------------------------------------
op nop {
  map nop:         0x00;  // not operate
  operands { - = { +0=0x00; }; };
};
//
op cp {
  //                          00...... = cp %,%
  map nop:         0x00;  // 00000000 = cp %I,%I  = nop
  map cp_i_j:      0x01;  // 00000001 = cp %I,%J
  map cp_i_a:      0x02;  // 00000010 = cp %I,%A
  map cp_i_b:      0x03;  // 00000011 = cp %I,%B
  map cp_i_bp:     0x04;  // 00000100 = cp %I,%BP
  map cp_i_sp:     0x05;  // 00000101 = cp %I,%SP
  map cp_i_c:      0x06;  // 00000110 = cp %I,%C
  map cp_i_fl:     0x07;  // 00000111 = cp %I,%FL
  map cp_j_i:      0x08;  // 00001000 = cp %J,%I
  map nop:         0x09;  // 00001001 = cp %J,%J  = nop
  map cp_j_a:      0x0A;  // 00001010 = cp %J,%A
  map cp_j_b:      0x0B;  // 00001011 = cp %J,%B
  map cp_j_bp:     0x0C;  // 00001100 = cp %J,%BP
  map cp_j_sp:     0x0D;  // 00001101 = cp %J,%SP
  map cp_j_c:      0x0E;  // 00001110 = cp %J,%C
  map cp_j_fl:     0x0F;  // 00001111 = cp %J,%FL
  map cp_a_i:      0x10;  // 00010000 = cp %A,%I
```

```
map cp_a_j:      0x11;  // 00010001 = cp %A,%J
map nop:         0x12;  // 00010010 = cp %A,%A  = nop
map cp_a_b:      0x13;  // 00010011 = cp %A,%B
map cp_a_bp:     0x14;  // 00010100 = cp %A,%BP
map cp_a_sp:     0x15;  // 00010101 = cp %A,%SP
map cp_a_c:      0x16;  // 00010110 = cp %A,%C
map cp_a_fl:     0x17;  // 00010111 = cp %A,%FL
map cp_b_i:      0x18;  // 00011000 = cp %B,%I
map cp_b_j:      0x19;  // 00011001 = cp %B,%J
map cp_b_a:      0x1A;  // 00011010 = cp %B,%A
map nop:         0x1B;  // 00011011 = cp %B,%B  = nop
map cp_b_bp:     0x1C;  // 00011100 = cp %B,%BP
map cp_b_sp:     0x1D;  // 00011101 = cp %B,%SP
map cp_b_c:      0x1E;  // 00011110 = cp %B,%C
map cp_b_fl:     0x1F;  // 00011111 = cp %B,%FL
map cp_bp_i:     0x20;  // 00100000 = cp %BP,%I
map cp_bp_j:     0x21;  // 00100001 = cp %BP,%J
map cp_bp_a:     0x22;  // 00100010 = cp %BP,%A
map cp_bp_b:     0x23;  // 00100011 = cp %BP,%B
map nop:         0x24;  // 00100100 = cp %BP,%BP = nop
map cp_bp_sp:    0x25;  // 00100101 = cp %BP,%SP
map cp_bp_c:     0x26;  // 00100110 = cp %BP,%C  = nop
map cp_bp_fl:    0x27;  // 00100111 = cp %BP,%FL
map cp_sp_i:     0x28;  // 00101000 = cp %SP,%I
map cp_sp_j:     0x29;  // 00101001 = cp %SP,%J
map cp_sp_a:     0x2A;  // 00101010 = cp %SP,%A
map cp_sp_b:     0x2B;  // 00101011 = cp %SP,%B
map cp_sp_bp:    0x2C;  // 00101100 = cp %SP,%BP
map nop:         0x2D;  // 00101101 = cp %SP,%SP = nop
map cp_sp_c:     0x2E;  // 00101110 = cp %SP,%C
map cp_sp_fl:    0x2F;  // 00101111 = cp %SP,%FL
map cp_c_i:      0x30;  // 00110000 = cp %C,%I
map cp_c_j:      0x31;  // 00110001 = cp %C,%J
map cp_c_a:      0x32;  // 00110010 = cp %C,%A
map cp_c_b:      0x33;  // 00110011 = cp %C,%B
map cp_c_bp:     0x34;  // 00110100 = cp %C,%BP
map cp_c_sp:     0x35;  // 00110101 = cp %C,%SP
map nop:         0x36;  // 00110110 = cp %C,%C   = nop
map cp_c_fl:     0x37;  // 00110111 = cp %C,%FL
map cp_fl_i:     0x38;  // 00111000 = cp %FL,%I
```

```
  map cp_fl_j:     0x39;  // 00111001 = cp %FL,%J
  map cp_fl_a:     0x3A;  // 00111010 = cp %FL,%A
  map cp_fl_b:     0x3B;  // 00111011 = cp %FL,%B
  map cp_fl_bp:    0x3C;  // 00111100 = cp %FL,%BP
  map cp_fl_sp:    0x3D;  // 00111101 = cp %FL,%SP
  map cp_fl_c:     0x3E;  // 00111110 = cp %FL,%C
  map nop:         0x3F;  // 00111111 = cp %FL,%FL = nop
  //                         010000.. = cp #,%
  map cp_num_i:    0x40;  // 01000000 = cp #,%I
  map cp_num_j:    0x41;  // 01000001 = cp #,%J
  map cp_num_a:    0x42;  // 01000010 = cp #,%A
  map cp_num_b:    0x43;  // 01000011 = cp #,%B
  //                         010001.. = cp (%),%
  map cp_mi_a:     0x44;  // 01000100 = cp (%I),%A
  map cp_mi_b:     0x45;  // 01000101 = cp (%I),%B
  map cp_mj_a:     0x46;  // 01000110 = cp (%J),%A
  map cp_mj_b:     0x47;  // 01000111 = cp (%J),%B
  //                         010010.. = cp %,%I
  map cp_a_mi:     0x48;  // 01001000 = cp %A,(%I)
  map cp_a_mj:     0x49;  // 01001001 = cp %A,(%J)
  map cp_b_mi:     0x4A;  // 01001010 = cp %B,(%I)
  map cp_b_mj:     0x4B;  // 01001011 = cp %B,(%J)
  //                         0100110. = cp #,(%)
  map cp_num_mi:   0x4C;  // 01001100 = cp #,(%I)
  map cp_num_mj:   0x4D;  // 01001101 = cp #,(%J)
  //                         0100111. = cp (%)
  map cp_mi_mj:    0x4E;  // 01001110 = cp (%I)
  map cp_mj_mi:    0x4F;  // 01001111 = cp (%J)
  //
  operands {
    %1,%2    = { +0=0x00; +0[5:3]=%1; +0[2:0]=%2; };
    #1,%2    = { +0=0x40; +0[1:0]=%2; +1=#1[7:0]; +2=#1[15:8]; };
    (%1),%2  = { +0=0x44; +0[1:1]=%1; +0[0:0]=%2; };
    %1,(%2)  = { +0=0x48; +0[1:1]=%1; +0[0:0]=%2; };
    #1,(%2)  = { +0=0x4C; +0[0:0]=%2; +1=#1[7:0]; +2=#1[15:8]; };
    (%1)     = { +0=0x4E; +0[0:0]=%1; };
  };
};
op cp8 {
  //                         010100.. = cp8 #,%
```

```
  map cp8_num_i:  0x50;  // 01010000 = cp8 #,%I
  map cp8_num_j:  0x51;  // 01010001 = cp8 #,%J
  map cp8_num_a:  0x52;  // 01010010 = cp8 #,%A
  map cp8_num_b:  0x53;  // 01010011 = cp8 #,%B
  //                        010101.. = cp8 (%),%
  map cp8_mi_a:   0x54;  // 01010100 = cp8 (%I),%A
  map cp8_mi_b:   0x55;  // 01010101 = cp8 (%I),%B
  map cp8_mj_a:   0x56;  // 01010110 = cp8 (%J),%A
  map cp8_mj_b:   0x57;  // 01010111 = cp8 (%J),%B
  //                        010110.. = cp8 %,(%)
  map cp8_a_mi:   0x58;  // 01011000 = cp8 %A,(%I)
  map cp8_a_mj:   0x59;  // 01011001 = cp8 %A,(%J)
  map cp8_b_mi:   0x5A;  // 01011010 = cp8 %B,(%I)
  map cp8_b_mj:   0x5B;  // 01011011 = cp8 %B,(%J)
  //                        0101110. = cp8 #,(%)
  map cp8_num_mi: 0x5C;  // 01011100 = cp8 #,(%I)
  map cp8_num_mj: 0x5D;  // 01011101 = cp8 #,(%J)
  //                        0101111. = cp8 (%)
  map cp8_mi_mj:  0x5E;  // 01011110 = cp8 (%I)
  map cp8_mj_mi:  0x5F;  // 01011111 = cp8 (%J)
  //
  operands {
    #1,%2    = { +0=0x50; +0[1:0]=%2; +1=#1[7:0]; };
    (%1),%2  = { +0=0x54; +0[1:1]=%1; +0[0:0]=%2; };
    %1,(%2)  = { +0=0x58; +0[1:1]=%1; +0[0:0]=%2; };
    #1,(%2)  = { +0=0x5C; +0[0:0]=%2; +1=#1[7:0]; };
    (%1)     = { +0=0x5E; +0[0:0]=%1; };
  };
};
//
op push {
  //                        01100... = push %
  map push_i:     0x60;  // 01100000 = push %I
  map push_j:     0x61;  // 01100001 = push %J
  map push_a:     0x62;  // 01100010 = push %A
  map push_b:     0x63;  // 01100011 = push %B
  map push_bp:    0x64;  // 01100100 = push %BP
  map push_sp:    0x65;  // 01100101 = push %SP
  map push_c:     0x66;  // 01100110 = push %C
  map push_fl:    0x67;  // 01100111 = push %FL
```

```
  //
  map push_num:    0x80;  // 10000000 = push #
  //
  map push_mi:     0x82;  // 10000010 = push (%I)
  map push_mj:     0x83;  // 10000011 = push (%J)
  //
  operands {
    %1    = { +0=0x60; +0[2:0]=%1; };
    #1    = { +0=0x80; +1=#1[7:0]; +2=#1[15:8]; };
    (%1)  = { +0=0x82; +0[0:0]=%1; };
  };
};
op pop {
  //                          01110... = pop %
  map pop_i:       0x70;  // 01110000 = pop %I
  map pop_j:       0x71;  // 01110001 = pop %J
  map pop_a:       0x72;  // 01110010 = pop %A
  map pop_b:       0x73;  // 01110011 = pop %B
  map pop_bp:      0x74;  // 01110100 = pop %BP
  map pop_sp:      0x75;  // 01110101 = pop %SP
  map pop_c:       0x76;  // 01110110 = pop %C
  map pop_fl:      0x77;  // 01110111 = pop %FL
  //                          1000011. = pop (%)
  map pop_mi:      0x86;  // 10000110 = pop (%I)
  map pop_mj:      0x87;  // 10000111 = pop (%J)
  //
  operands {
    %1    = { +0=0x70; +0[2:0]=%1; };
    (%1)  = { +0=0x86; +0[0:0]=%1; };
  };
};
//
op push8 {
  //                          01101... = push8 %
  map push8_i:     0x68;  // 01101000 = push8 %I
  map push8_j:     0x69;  // 01101001 = push8 %J
  map push8_a:     0x6A;  // 01101010 = push8 %A
  map push8_b:     0x6B;  // 01101011 = push8 %B
  map push8_bp:    0x6C;  // 01101100 = push8 %BP
  map push8_sp:    0x6D;  // 01101101 = push8 %SP
```

```
  map push8_c:     0x6E;  // 01101110 = push8 %C
  map push8_fl:    0x6F;  // 01101111 = push8 %FL
  //
  map push8_num:   0x81;  // 10000001 = push8 #
  //                         1000010. = push8 (%I)
  map push8_mi:    0x84;  // 10000100 = push8 (%I)
  map push8_mj:    0x85;  // 10000101 = push8 (%J)
  //
  operands {
    %1    = { +0=0x68; +0[2:0]=%1; };
    #1    = { +0=0x81; +1[7:0]=#1; };
    (%1)  = { +0=0x81; +0[0:0]=%1; };
  };
};
op pop8 {
  //                         01111... = pop8 %
  map pop8_i:      0x78;  // 01111000 = pop8 %I
  map pop8_j:      0x79;  // 01111001 = pop8 %J
  map pop8_a:      0x7A;  // 01111010 = pop8 %A
  map pop8_b:      0x7B;  // 01111011 = pop8 %B
  map pop8_bp:     0x7C;  // 01111100 = pop8 %BP
  map pop8_sp:     0x7D;  // 01111101 = pop8 %SP
  map pop8_c:      0x7E;  // 01111110 = pop8 %C
  map pop8_fl:     0x7F;  // 01111111 = pop8 %FL
  //                       // 1000100. = pop8 (%)
  map pop8_mi:     0x88;  // 10001000 = pop8 (%I)
  map pop8_mj:     0x89;  // 10001001 = pop8 (%J)
  //
  operands {
    %1    = { +0=0x78; +0[2:0]=%1; };
    (%1)  = { +0=0x88; +0[0:0]=%1; };
  };
};
//
op jump {
  map jump_nc:     0x90;  // 10010000 = jump %carry 0 #
  map jump_c:      0x91;  // 10010001 = jump %carry 1 #
  map jump_nz:     0x92;  // 10010010 = jump %zero  0 #
  map jump_z:      0x93;  // 10010011 = jump %zero  1 #
  map jump_nn:     0x94;  // 10010100 = jump %negative 0 #
```

```
  map jump_n:      0x95;  // 10010101 = jump %negative 1 #
  map jump_no:     0x96;  // 10010110 = jump %overflow 0 #
  map jump_o:      0x97;  // 10010111 = jump %overflow 1 #
  //
  map jump:        0xFE;  // 11111110 = jump #
  //
  operands {
     %1,#2,#3 = { +0=0x90; +0[2:1]=%1; +0[0:0]=#2;
                  +1=#3[7:0]; +2=#3[15:8]; };
     #1       = { +0=0xFE; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
//
op equal {
  map equal:       0x98;  // 10011000 = equal
  operands { - = { +0=0x98; }; };
};
op not {
  map not:         0x99;  // 10011001 = not
  operands { - = { +0=0x99; }; };
};
//
op and {
  map and:         0x9A;  // 10011010 = and
  operands { - = { +0=0x9A; }; };
};
op nand {
  map nand:        0x9B;  // 10011011 = nand
  operands { - = { +0=0x9B; }; };
};
//
op or {
  map or:          0x9C;  // 10011100 = or
  operands { - = { +0=0x9C; }; };
};
op nor {
  map nor:         0x9D;  // 10011101 = nor
  operands { - = { +0=0x9D; }; };
};
//
```

```
op xor {
  map xor:          0x9E;  // 10011110 = xor
  operands { - = { +0=0x9E; }; };
};
op nxor {
  map nxor:         0x9F;  // 10011111 = nxor
  operands { - = { +0=0x9F; }; };
};
//
op add {
  map add:          0xA0;  // 10100000 = add
  operands { - = { +0=0xA0; }; };
};
op sub {
  map sub:          0xA1;  // 10100001 = sub
  operands { - = { +0=0xA1; }; };
};
//
op addc {
  map addc:         0xA2;  // 10100010 = addc
  operands { - = { +0=0xA2; }; };
};
op subb {
  map subb:         0xA3;  // 10100011 = subb
  operands { - = { +0=0xA3; }; };
};
//
op lshl {
  map lshl:         0xA4;  // 10100100 = lshl
  operands { - = { +0=0xA4; }; };
};
op lshr {
  map lshr:         0xA5;  // 10100101 = lshr
  operands { - = { +0=0xA5; }; };
};
//
op ashl {
  map ashl:         0xA6;  // 10100110 = ashl
  operands { - = { +0=0xA6; }; };
};
```

```
op ashr {
  map ashr:        0xA7;  // 10100111 = ashr
  operands { - = { +0=0xA7; }; };
};
//
op rotcl {
  map rotcl:       0xA8;  // 10101000 = rotcl
  operands { - = { +0=0xA8; }; };
};
op rotcr {
  map rotcr:       0xA9;  // 10101001 = rotcr
  operands { - = { +0=0xA9; }; };
};
//
op rotl {
  map rotl:        0xAA;  // 10101010 = rotl
  operands { - = { +0=0xAA; }; };
};
op rotr {
  map rotr:        0xAB;  // 10101011 = rotr
  operands { - = { +0=0xAB; }; };
};
//
op add8 {
  map add8:        0xAC;  // 10101100 = add8
  operands { - = { +0=0xAC; }; };
};
op sub8 {
  map sub8:        0xAD;  // 10101101 = sub8
  operands { - = { +0=0xAD; }; };
};
//
op add8c {
  map add8c:       0xAE;  // 10101110 = addc8
  operands { - = { +0=0xAE; }; };
};
op sub8b {
  map sub8b:       0xAF;  // 10101111 = subb8
  operands { - = { +0=0xAF; }; };
};
```

```
//
op lsh8l {
  map lsh8l:      0xB0;  // 10110000 = lsh18l
  operands { - = { +0=0xB0; }; };
};
op lsh8r {
  map lsh8r:      0xB1;  // 10110001 = lsh18r
  operands { - = { +0=0xB1; }; };
};
//
op ash8l {
  map ash8l:      0xB2;  // 10110010 = ash18l
  operands { - = { +0=0xB2; }; };
};
op ash8r {
  map ash8r:      0xB3;  // 10110011 = ash18r
  operands { - = { +0=0xB3; }; };
};
//
op rot8cl {
  map rot8cl:     0xB4;  // 10110100 = rotc81
  operands { - = { +0=0xB4; }; };
};
op rot8cr {
  map rot8cr:     0xB5;  // 10110101 = rotc8r
  operands { - = { +0=0xB5; }; };
};
//
op rot8l {
  map rot8l:      0xB6;  // 10110110 = rot81
  operands { - = { +0=0xB6; }; };
};
op rot8r {
  map rot8r:      0xB7;  // 10110111 = rot8r
  operands { - = { +0=0xB7; }; };
};
//
op cast {
  //                    // 101110.. = cast <sig>,%
  map cast_uns_a: 0xB8;  // 10111000 = cast 0,%A
```

```
  map cast_uns_b: 0xB9;  // 10111001 = cast 0,%B
  map cast_sig_a: 0xBA;  // 10111010 = cast 1,%A
  map cast_sig_b: 0xBB;  // 10111011 = cast 1,%B
  operands {
    #1,%2 = {+0=0xB8; +0[1:1]=#1; +0[0:0]=%2; };
  };
};
//
op equals {
  map equals:     0xBC;  // 10111100 = stack <= %A
  operands { - = { +0=0xBC; }; };
};
op nots {
  map nots:       0xBD;  // 10111101 = stack <= NOT %A
  operands { - = { +0=0xBD; }; };
};
//
op ands {
  map ands:       0xBE;  // 10111110 = stack <= %A AND %B
  operands { - = { +0=0xBE; }; };
};
op nands {
  map nands:      0xBF;  // 10111111 = stack <= %A NAND %B
  operands { - = { +0=0xBF; }; };
};
//
op ors {
  map ors:        0xC0;  // 11000000 = stack <= %A OR %B
  operands { - = { +0=0xC0; }; };
};
op nors {
  map nors:       0xC1;  // 11000001 = stack <= %A NOR %B
  operands { - = { +0=0xC1; }; };
};
//
op xors {
  map xors:       0xC2;  // 11000010 = stack <= %A XOR %B
  operands { - = { +0=0xC2; }; };
};
op nxors {
```

```
  map nxors:       0xC3;  // 11000011 = stack <= %A NXOR %B
  operands { - = { +0=0xC3; }; };
};
//
op adds {
  map adds:        0xC4;  // 11000100 = stack <= %A + %B
  operands { - = { +0=0xC4; }; };
};
op subs {
  map subs:        0xC5;  // 11000101 = stack <= %A - %B
  operands { - = { +0=0xC5; }; };
};
//
op addcs {
  map addcs:       0xC6;  // 11000110 = stack <= %A + %B + %carry
  operands { - = { +0=0xC6; }; };
};
op subbs {
  map subbs:       0xC7;  // 11000111 = stack <= %A - %B - %carry
  operands { - = { +0=0xC7; }; };
};
//
op lshls {
  map lshls:       0xC8;  // 11001000 = stack <= lshl
  operands { - = { +0=0xC8; }; };
};
op lshrs {
  map lshrs:       0xC9;  // 11001001 = stack <= lshr
  operands { - = { +0=0xC9; }; };
};
//
op ashls {
  map ashls:       0xCA;  // 11001010 = stack <= ashl
  operands { - = { +0=0xCA; }; };
};
op ashrs {
  map ashrs:       0xCB;  // 11001011 = stack <= ashr
  operands { - = { +0=0xCB; }; };
};
//
```

```
op rotcls {
  map rotcls:      0xCC;  // 11001100 = stack <= rotcl
  operands { - = { +0=0xCC; }; };
};
op rotcrs {
  map rotcrs:      0xCD;  // 11001101 = stack <= rotcr
  operands { - = { +0=0xCD; }; };
};
//
op rotls {
  map rotls:       0xCE;  // 11001110 = stack <= rotl
  operands { - = { +0=0xCE; }; };
};
op rotrs {
  map rotrs:       0xCF;  // 11001111 = stack <= rotr
  operands { - = { +0=0xCF; }; };
};
//
op add8s {
  map add8s:       0xD0;  // 11010000 = stack8 <= +
  operands { - = { +0=0xD0; }; };
};
op sub8s {
  map sub8s:       0xD1;  // 11010001 = stack8 <= -
  operands { - = { +0=0xD1; }; };
};
//
op add8cs {
  map add8cs:      0xD2;  // 11010010 = stack8 <= + + %carry
  operands { - = { +0=0xD2; }; };
};
op sub8bs {
  map sub8bs:      0xD3;  // 11010011 = stack8 <= - - %carry
  operands { - = { +0=0xD3; }; };
};
//
op lsh8ls {
  map lsh8ls:      0xD4;  // 11010100 = stack8 <= lsh8r
  operands { - = { +0=0xD4; }; };
};
```

```
op lsh8rs {
  map lsh8rs:     0xD5;  // 11010101 = stack8 <= lsh8r
  operands { - = { +0=0xD5; }; };
};
//
op ash8ls {
  map ash8ls:     0xD6;  // 11010110 = stack8 <= ash8l
  operands { - = { +0=0xD6; }; };
};
op ash8rs {
  map ash8rs:     0xD7;  // 11010111 = stack8 <= ash8r
  operands { - = { +0=0xD7; }; };
};
//
op rot8cls {
  map rot8cls:    0xD8;  // 11011000 = stack8 <= rot8cl
  operands { - = { +0=0xD8; }; };
};
op rot8crs {
  map rot8crs:    0xD9;  // 11011001 = stack8 <= rot8cr
  operands { - = { +0=0xD9; }; };
};
//
op rot8ls {
  map rot8ls:     0xDA;  // 11011010 = stack8 <= rot8l
  operands { - = { +0=0xDA; }; };
};
op rot8rs {
  map rot8rs:     0xDB;  // 11011011 = stack8 <= rot8r
  operands { - = { +0=0xDB; }; };
};
//
op casts {
  //                    // 1101110. = casts <sig>
  map casts_uns:  0xDC;  // 11011100 = casts 0
  map casts_sig:  0xDD;  // 11011101 = casts 1
  operands {
    #1 = {+0=0xDC; +0[0:0]=#1; };
  };
};
```

```
//
op call {
  map call:       0xDE;  // 11011110 = call #
  map call_i:     0xFC;  // 11111100 = call (%I)
  map call_j:     0xFD;  // 11111101 = call (%J)
  operands {
    #1   = { +0=0xDE; +1=#1[7:0]; +2=#1[15:8]; };
    (%1) = { +0=0xFC; +0[0:0]=%1; };
  };
};
op return {
  map return:     0xDF;  // 11011111 = return
  operands { - = { +0=0xDF; }; };
};
//
op int {
  map int:        0xE0;  // 11100000 = int <n>
  operands {
    #1 = { +0=0xE0; +1=#1; };
  };
};
op iret {
  map iret:       0xE1;  // 11100001 = iret
  operands { - = { +0=0xE1; }; };
};
//
op imrl {
  map imrl:       0xE2;  // 11100010 = IMR load
  operands {
    #1   = { +0=0xE2; +1=#1[7:0]; };
  };
  operands { - = { +0=0xE2; }; };
};
op ivtl {
  map ivtl:       0xE3;  // 11100011 = IVT load
  operands {
    #1   = { +0=0xE3; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
//
```

```
op flag_i {
  map flag_i_clr: 0xE4;  // 11100100 = flag_i 0
  map flag_i_set: 0xE5;  // 11100101 = flag_i 1
  operands {
    #1 = { +0=0xE4; +0[0:0]=#1; };
  };
};
//
op flag_c {
  map flag_c_clr: 0xE6;  // 11100110 = flag_c 0
  map flag_c_set: 0xE7;  // 11100111 = flag_c 1
  operands {
    #1 = { +0=0xE6; +0[0:0]=#1; };
  };
};
//
op in {
  map in_num_a:   0xE8;  // 11101000 = in <n>,%A
  map in_num_b:   0xE9;  // 11101001 = in <n>,%B
  map in_num_mi:  0xEA;  // 11101010 = in <n>,(%I)
  map in_num_mj:  0xEB;  // 11101011 = in <n>,(%J)
  operands {
    #1,%2   = { +0=0xE8; +0[0:0]=%2; +1=#1; };
    #1,(%2) = { +0=0xEA; +0[0:0]=%2; +1=#1; };
  };
};
op out {
  map out_a_num:  0xEC;  // 11101100 = out %A,#
  map out_b_num:  0xED;  // 11101101 = out %B,#
  map out_mi_num: 0xEE;  // 11101110 = out (%I),#
  map out_mi_num: 0xEF;  // 11101111 = out (%J),#
  operands {
    %1,#2   = { +0=0xEC; +0[0:0]=%1; +1=#2; };
    (%1),#2 = { +0=0xEE; +0[0:0]=%1; +1=#2; };
  };
};
//
op ifack_jump {
  map ifack_jump: 0xF0;  // 11110000 = jump if ack from device
  operands {
```

```
    #1,#2    = { +0=0xF0; +1=#1; +2=#2[7:0]; +3=#2[15:8]; };
  };
};
op ifack_call {
  map ifack_call: 0xF1;  // 11110001 = call if ack from device
  operands {
    #1,#2    = { +0=0xF1; +1=#1; +2=#2[7:0]; +3=#2[15:8]; };
  };
};
//
op cmpr {
  map cmpr:        0xF2;  // 11110010 = compare %A, %B
  operands { - = { +0[7:0]=0xF2; }; };
};
op testr {
  map testr:       0xF3;  // 11110011 = test %A, %B
  operands { - = { +0[7:0]=0xF3; }; };
};
//
op cmpi {
  map cmpi:        0xF4;  // 11110100 = compare (%I), (%J)
  operands { - = { +0[7:0]=0xF4; }; };
};
op testi {
  map testi:       0xF5;  // 11110101 = test (%I), (%J)
  operands { - = { +0[7:0]=0xF5; }; };
};
//
op cmps {
  map cmps:        0xF6;  // 11110110 = compare stack
  operands { - = { +0[7:0]=0xF6; }; };
};
op tests {
  map tests:       0xF7;  // 11110111 = test stack
  operands { - = { +0[7:0]=0xF7; }; };
};
//
op cmp8i {
  map cmp8i:       0xF8;  // 11111000 = compare8 (%I), (%J)
  operands { - = { +0[7:0]=0xF8; }; };
```

```
};
op test8i {
  map test8i:      0xF9;  // 11111001 = test8 (%I), (%J)
  operands { - = { +0[7:0]=0xF9; }; };
};
//
op cmp8s {
  map cmp8s:       0xFA;  // 11111010 = compare8 stack
  operands { - = { +0[7:0]=0xFA; }; };
};
op test8s {
  map test8s:      0xFB;  // 11111011 = test8 stack
  operands { - = { +0[7:0]=0xFB; }; };
};
//
op stop {
  map stop:       0xFF;  // 11111111
  operands { - = { +0[7:0]=0xFF; }; };
};
```

11.15 Microcode

It follows the microcode listing, describing the opcode procedures.

Listing 11.43. Microcode for TKGate.

```
begin microcode @ 0
//
fetch:
    // IR[7:0] <= RAM[PC++], MPC <= IR[7:0]
    ir=select_bus_b ir=rank_8 ir=register_load_byte_1 ir=select_byte_1
    ram=select_bus_b ram=bus_write ram=p
    pc=dp1 ctrl=load;
//
nop:
    // MPC <= #fetch
    addr=fetch jump=true;
//
cp_i_j:
    // J <= I, MPC <= #fetch
    j=select_bus_a j=rank_16 j=register_load
```

```
    i=select_bus_a i=rank_16 i=bus_write
    addr=fetch jump=true;
cp_i_a:
    // A <= I, MPC <= #fetch
    a=select_bus_a a=rank_16 a=register_load
    i=select_bus_a i=rank_16 i=bus_write
    addr=fetch jump=true;
cp_i_b:
    // B <= I, MPC <= #fetch
    b=select_bus_a b=rank_16 b=register_load
    i=select_bus_a i=rank_16 i=bus_write
    addr=fetch jump=true;
cp_i_sp:
    // SP <= I, MPC <= #fetch
    sp=select_bus_a sp=rank_16 sp=register_load
    i=select_bus_a i=rank_16 i=bus_write
    addr=fetch jump=true;
cp_i_bp:
    // BP <= I, MPC <= #fetch
    bp=select_bus_a bp=rank_16 bp=register_load
    i=select_bus_a i=rank_16 i=bus_write
    addr=fetch jump=true;
cp_i_c:
    // C <= I, MPC <= #fetch
    c=select_bus_a c=rank_16 c=register_load
    i=select_bus_a i=rank_16 i=bus_write
    addr=fetch jump=true;
cp_i_fl:
    // FL <= I, MPC <= #fetch
    fl=select_bus_a fl=rank_16 fl=register_load
    i=select_bus_a i=rank_16 i=bus_write
    addr=fetch jump=true;
cp_j_i:
    // I <= J, MPC <= #fetch
    i=select_bus_a i=rank_16 i=register_load
    j=select_bus_a j=rank_16 j=bus_write
    addr=fetch jump=true;
cp_j_a:
    // A <= J, MPC <= #fetch
    a=select_bus_a a=rank_16 a=register_load
    j=select_bus_a j=rank_16 j=bus_write
```

```
    addr=fetch jump=true;
cp_j_b:
    // B <= J, MPC <= #fetch
    b=select_bus_a b=rank_16 b=register_load
    j=select_bus_a j=rank_16 j=bus_write
    addr=fetch jump=true;
cp_j_sp:
    // SP <= J, MPC <= #fetch
    sp=select_bus_a sp=rank_16 sp=register_load
    j=select_bus_a j=rank_16 j=bus_write
    addr=fetch jump=true;
cp_j_bp:
    // BP <= J, MPC <= #fetch
    bp=select_bus_a bp=rank_16 bp=register_load
    j=select_bus_a j=rank_16 j=bus_write
    addr=fetch jump=true;
cp_j_c:
    // C <= J, MPC <= #fetch
    c=select_bus_a c=rank_16 c=register_load
    j=select_bus_a j=rank_16 j=bus_write
    addr=fetch jump=true;
cp_j_fl:
    // FL <= J, MPC <= #fetch
    fl=select_bus_a fl=rank_16 fl=register_load
    j=select_bus_a j=rank_16 j=bus_write
    addr=fetch jump=true;
cp_a_i:
    // I <= A, MPC <= #fetch
    i=select_bus_a i=rank_16 i=register_load
    a=select_bus_a a=rank_16 a=bus_write
    addr=fetch jump=true;
cp_a_j:
    // J <= A, MPC <= #fetch
    j=select_bus_a j=rank_16 j=register_load
    a=select_bus_a a=rank_16 a=bus_write
    addr=fetch jump=true;
cp_a_b:
    // B <= A, MPC <= #fetch
    b=select_bus_a b=rank_16 b=register_load
    a=select_bus_a a=rank_16 a=bus_write
    addr=fetch jump=true;
```

```
cp_a_sp:
    // SP <= A, MPC <= #fetch
    sp=select_bus_a sp=rank_16 sp=register_load
    a=select_bus_a a=rank_16 a=bus_write
    addr=fetch jump=true;
cp_a_bp:
    // BP <= A, MPC <= #fetch
    bp=select_bus_a bp=rank_16 bp=register_load
    a=select_bus_a a=rank_16 a=bus_write
    addr=fetch jump=true;
cp_a_c:
    // C <= A, MPC <= #fetch
    c=select_bus_a c=rank_16 c=register_load
    a=select_bus_a a=rank_16 a=bus_write
    addr=fetch jump=true;
cp_a_fl:
    // FL <= A, MPC <= #fetch
    fl=select_bus_a fl=rank_16 fl=register_load
    a=select_bus_a a=rank_16 a=bus_write
    addr=fetch jump=true;
cp_b_i:
    // I <= B, MPC <= #fetch
    i=select_bus_a i=rank_16 i=register_load
    b=select_bus_a b=rank_16 b=bus_write
    addr=fetch jump=true;
cp_b_j:
    // J <= B, MPC <= #fetch
    j=select_bus_a j=rank_16 j=register_load
    b=select_bus_a b=rank_16 b=bus_write
    addr=fetch jump=true;
cp_b_a:
    // A <= B, MPC <= #fetch
    a=select_bus_a a=rank_16 a=register_load
    b=select_bus_a b=rank_16 b=bus_write
    addr=fetch jump=true;
cp_b_sp:
    // SP <= B, MPC <= #fetch
    sp=select_bus_a sp=rank_16 sp=register_load
    b=select_bus_a b=rank_16 b=bus_write
    addr=fetch jump=true;
cp_b_bp:
```

```
    // BP <= B, MPC <= #fetch
    bp=select_bus_a bp=rank_16 bp=register_load
    b=select_bus_a b=rank_16 b=bus_write
    addr=fetch jump=true;
cp_b_c:
    // C <= B, MPC <= #fetch
    c=select_bus_a c=rank_16 c=register_load
    b=select_bus_a b=rank_16 b=bus_write
    addr=fetch jump=true;
cp_b_fl:
    // FL <= B, MPC <= #fetch
    fl=select_bus_a fl=rank_16 fl=register_load
    b=select_bus_a b=rank_16 b=bus_write
    addr=fetch jump=true;
cp_sp_i:
    // I <= SP, MPC <= #fetch
    i=select_bus_a i=rank_16 i=register_load
    sp=select_bus_a sp=rank_16 sp=bus_write
    addr=fetch jump=true;
cp_sp_j:
    // J <= SP, MPC <= #fetch
    j=select_bus_a j=rank_16 j=register_load
    sp=select_bus_a sp=rank_16 sp=bus_write
    addr=fetch jump=true;
cp_sp_a:
    // A <= SP, MPC <= #fetch
    a=select_bus_a a=rank_16 a=register_load
    sp=select_bus_a sp=rank_16 sp=bus_write
    addr=fetch jump=true;
cp_sp_b:
    // B <= SP, MPC <= #fetch
    b=select_bus_a b=rank_16 b=register_load
    sp=select_bus_a sp=rank_16 sp=bus_write
    addr=fetch jump=true;
cp_sp_bp:
    // BP <= SP, MPC <= #fetch
    bp=select_bus_a bp=rank_16 bp=register_load
    sp=select_bus_a sp=rank_16 sp=bus_write
    addr=fetch jump=true;
cp_sp_c:
    // C <= SP, MPC <= #fetch
```

```
    c=select_bus_a c=rank_16 c=register_load
    sp=select_bus_a sp=rank_16 sp=bus_write
    addr=fetch jump=true;
cp_sp_fl:
    // FL <= SP, MPC <= #fetch
    fl=select_bus_a fl=rank_16 fl=register_load
    sp=select_bus_a sp=rank_16 sp=bus_write
    addr=fetch jump=true;
cp_bp_i:
    // I <= BP, MPC <= #fetch
    i=select_bus_a i=rank_16 i=register_load
    bp=select_bus_a bp=rank_16 bp=bus_write
    addr=fetch jump=true;
cp_bp_j:
    // J <= BP, MPC <= #fetch
    j=select_bus_a j=rank_16 j=register_load
    bp=select_bus_a bp=rank_16 bp=bus_write
    addr=fetch jump=true;
cp_bp_a:
    // A <= BP, MPC <= #fetch
    a=select_bus_a a=rank_16 a=register_load
    bp=select_bus_a bp=rank_16 bp=bus_write
    addr=fetch jump=true;
cp_bp_b:
    // B <= BP, MPC <= #fetch
    b=select_bus_a b=rank_16 b=register_load
    bp=select_bus_a bp=rank_16 bp=bus_write
    addr=fetch jump=true;
cp_bp_sp:
    // SP <= BP, MPC <= #fetch
    sp=select_bus_a sp=rank_16 sp=register_load
    bp=select_bus_a bp=rank_16 bp=bus_write
    addr=fetch jump=true;
cp_bp_c:
    // C <= BP, MPC <= #fetch
    c=select_bus_a c=rank_16 c=register_load
    bp=select_bus_a bp=rank_16 bp=bus_write
    addr=fetch jump=true;
cp_bp_fl:
    // FL <= BP, MPC <= #fetch
    fl=select_bus_a fl=rank_16 fl=register_load
```

```
    bp=select_bus_a bp=rank_16 bp=bus_write
    addr=fetch jump=true;
cp_c_i:
    // I <= C, MPC <= #fetch
    i=select_bus_a i=rank_16 i=register_load
    c=select_bus_a c=rank_16 c=bus_write
    addr=fetch jump=true;
cp_c_j:
    // J <= C, MPC <= #fetch
    j=select_bus_a j=rank_16 j=register_load
    c=select_bus_a c=rank_16 c=bus_write
    addr=fetch jump=true;
cp_c_a:
    // A <= C, MPC <= #fetch
    a=select_bus_a a=rank_16 a=register_load
    c=select_bus_a c=rank_16 c=bus_write
    addr=fetch jump=true;
cp_c_b:
    // B <= C, MPC <= #fetch
    b=select_bus_a b=rank_16 b=register_load
    c=select_bus_a c=rank_16 c=bus_write
    addr=fetch jump=true;
cp_c_sp:
    // SP <= C, MPC <= #fetch
    sp=select_bus_a sp=rank_16 sp=register_load
    c=select_bus_a c=rank_16 c=bus_write
    addr=fetch jump=true;
cp_c_bp:
    // BP <= C, MPC <= #fetch
    bp=select_bus_a bp=rank_16 bp=register_load
    c=select_bus_a c=rank_16 c=bus_write
    addr=fetch jump=true;
cp_c_fl:
    // FL <= C, MPC <= #fetch
    fl=select_bus_a fl=rank_16 fl=register_load
    c=select_bus_a c=rank_16 c=bus_write
    addr=fetch jump=true;
cp_fl_i:
    // I <= FL, MPC <= #fetch
    i=select_bus_a i=rank_16 i=register_load
    fl=select_bus_a fl=rank_16 fl=bus_write
```

```
    addr=fetch jump=true;
cp_fl_j:
    // J <= FL, MPC <= #fetch
    j=select_bus_a j=rank_16 j=register_load
    fl=select_bus_a fl=rank_16 fl=bus_write
    addr=fetch jump=true;
cp_fl_a:
    // A <= FL, MPC <= #fetch
    a=select_bus_a a=rank_16 a=register_load
    fl=select_bus_a fl=rank_16 fl=bus_write
    addr=fetch jump=true;
cp_fl_b:
    // B <= FL, MPC <= #fetch
    b=select_bus_a b=rank_16 b=register_load
    fl=select_bus_a fl=rank_16 fl=bus_write
    addr=fetch jump=true;
cp_fl_sp:
    // SP <= FL, MPC <= #fetch
    sp=select_bus_a sp=rank_16 sp=register_load
    fl=select_bus_a fl=rank_16 fl=bus_write
    addr=fetch jump=true;
cp_fl_bp:
    // BP <= FL, MPC <= #fetch
    bp=select_bus_a bp=rank_16 bp=register_load
    fl=select_bus_a fl=rank_16 fl=bus_write
    addr=fetch jump=true;
cp_fl_c:
    // C <= FL, MPC <= #fetch
    c=select_bus_a c=rank_16 c=register_load
    fl=select_bus_a fl=rank_16 fl=bus_write
    addr=fetch jump=true;
cp_num_i:
    // I[7:0] <= RAM[PC++]
    i=select_bus_a i=rank_8 i=register_load_byte_1 i=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // I[15:8] <= RAM[PC++], MPC <= #fetch
    i=select_bus_a i=rank_8 i=register_load_byte_2 i=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1
    addr=fetch jump=true;
cp_num_j:
    // J[7:0] <= RAM[PC++]
```

```
    j=select_bus_a j=rank_8 j=register_load_byte_1 j=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // J[15:8] <= RAM[PC++], MPC <= #fetch
    j=select_bus_a j=rank_8 j=register_load_byte_2 j=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1
    addr=fetch jump=true;
cp_num_a:
    // A[7:0] <= RAM[PC++]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // A[15:8] <= RAM[PC++], MPC <= #fetch
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1
    addr=fetch jump=true;
cp_num_b:
    // B[7:0] <= RAM[PC++]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // B[15:8] <= RAM[PC++], MPC <= #fetch
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1
    addr=fetch jump=true;
cp_mi_a:
    // A[7:0] <= RAM[I]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=i;
    // A[15:8] <= RAM[I+1], MPC <= #fetch
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=i i=qp1
    addr=fetch jump=true;
cp_mi_b:
    // B[7:0] <= RAM[I]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=i;
    // B[15:8] <= RAM[I+1], MPC <= #fetch
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=i i=qp1
    addr=fetch jump=true;
cp_mj_a:
    // A[7:0] <= RAM[J]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
```

```
    ram=select_bus_a ram=bus_write ram=j;
    // A[15:8] <= RAM[J+1], MPC <= #fetch
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=j j=qp1
    addr=fetch jump=true;
cp_mj_b:
    // B[7:0] <= RAM[J]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=j;
    // B[15:8] <= RAM[J+1], MPC <= #fetch
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=j j=qp1
    addr=fetch jump=true;
cp_a_mi:
    // RAM[I] <= A[7:0]
    ram=select_bus_a ram=load ram=i
    a=select_bus_a a=rank_8 a=select_byte_1 a=bus_write;
    // RAM[I+1] <= A[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=i i=qp1
    a=select_bus_a a=rank_8 a=select_byte_2 a=bus_write
    addr=fetch jump=true;
cp_a_mj:
    // RAM[J] <= A[7:0]
    ram=select_bus_a ram=load ram=j
    a=select_bus_a a=rank_8 a=select_byte_1 a=bus_write;
    // RAM[J+1] <= A[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=j j=qp1
    a=select_bus_a a=rank_8 a=select_byte_2 a=bus_write
    addr=fetch jump=true;
cp_b_mi:
    // RAM[I] <= B[7:0]
    ram=select_bus_a ram=load ram=i
    b=select_bus_a b=rank_8 b=select_byte_1 b=bus_write;
    // RAM[I+1] <= B[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=i i=qp1
    b=select_bus_a b=rank_8 b=select_byte_2 b=bus_write
    addr=fetch jump=true;
cp_b_mj:
    // RAM[J] <= B[7:0]
    ram=select_bus_a ram=load ram=j
    b=select_bus_a b=rank_8 b=select_byte_1 b=bus_write;
```

```
    // RAM[J+1] <= B[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=j j=qp1
    b=select_bus_a b=rank_8 b=select_byte_2 b=bus_write
    addr=fetch jump=true;
cp_num_mi:
    // TMP[7:0] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // TMP[15:8] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // RAM[I] <= TMP[7:0]
    ram=select_bus_a ram=load ram=i
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_1 tmp=bus_write;
    // RAM[I+1] <= TMP[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=i i=qp1
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_2 tmp=bus_write
    addr=fetch jump=true;
cp_num_mj:
    // TMP[7:0] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // TMP[15:8] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // RAM[J] <= TMP[7:0]
    ram=select_bus_a ram=load ram=j
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_1 tmp=bus_write;
    // RAM[J+1] <= TMP[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=j j=qp1
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_2 tmp=bus_write
    addr=fetch jump=true;
cp_mi_mj:
    // TMP[7:0] <= RAM[I]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=i;
    // TMP[15:8] <= RAM[I+1]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=i i=qp1;
    // RAM[J] <= TMP[7:0]
    ram=select_bus_a ram=load ram=j
```

```
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_1 tmp=bus_write;
    // RAM[J+1] <= TMP[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=j j=qp1
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_2 tmp=bus_write
    addr=fetch jump=true;
cp_mj_mi:
    // TMP[7:0] <= RAM[J]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=j;
    // TMP[15:8] <= RAM[J+1]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=j j=qp1;
    // RAM[I] <= TMP[7:0]
    ram=select_bus_a ram=load ram=i
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_1 tmp=bus_write;
    // RAM[I+1] <= TMP[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=i i=qp1
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_2 tmp=bus_write
    addr=fetch jump=true;
cp8_num_i:
    // I <= RAM[PC++], MPC <= #fetch
    i=select_bus_a i=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1
    addr=fetch jump=true;
cp8_num_j:
    // J <= RAM[PC++], MPC <= #fetch
    j=select_bus_a j=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1
    addr=fetch jump=true;
cp8_num_a:
    // A <= RAM[PC++], MPC <= #fetch
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1
    addr=fetch jump=true;
cp8_num_b:
    // B <= RAM[PC++], MPC <= #fetch
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1
    addr=fetch jump=true;
cp8_mi_a:
    // A <= RAM[I], MPC <= #fetch
```

```
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=i
    addr=fetch jump=true;
cp8_mi_b:
    // B <= RAM[I], MPC <= #fetch
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=i
    addr=fetch jump=true;
cp8_mj_a:
    // A <= RAM[J], MPC <= #fetch
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=j
    addr=fetch jump=true;
cp8_mj_b:
    // B <= RAM[J], MPC <= #fetch
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=j
    addr=fetch jump=true;
cp8_a_mi:
    // RAM[I] <= B, MPC <= #fetch
    ram=select_bus_a ram=load ram=i
    a=select_bus_a a=bus_write
    addr=fetch jump=true;
cp8_a_mj:
    // RAM[J] <= B, MPC <= #fetch
    ram=select_bus_a ram=load ram=j
    a=select_bus_a a=bus_write
    addr=fetch jump=true;
cp8_b_mi:
    // RAM[I] <= B, MPC <= #fetch
    ram=select_bus_a ram=load ram=i
    b=select_bus_a b=bus_write
    addr=fetch jump=true;
cp8_b_mj:
    // RAM[J] <= B, MPC <= #fetch
    ram=select_bus_a ram=load ram=j
    b=select_bus_a b=bus_write
    addr=fetch jump=true;
cp8_num_mi:
    // TMP <= RAM[PC++]
    tmp=select_bus_a tmp=register_load
```

```
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // RAM[I] <= TMP, MPC <= #fetch
    ram=select_bus_a ram=load ram=i
    tmp=select_bus_a tmp=bus_write
    addr=fetch jump=true;
cp8_num_mj:
    // TMP <= RAM[PC++]
    tmp=select_bus_a tmp=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // RAM[J] <= TMP, MPC <= #fetch
    ram=select_bus_a ram=load ram=j
    tmp=select_bus_a tmp=bus_write
    addr=fetch jump=true;
cp8_mi_mj:
    // TMP <= RAM[I]
    tmp=select_bus_a tmp=register_load
    ram=select_bus_a ram=bus_write ram=i;
    // RAM[J] <= TMP, MPC <= #fetch
    ram=select_bus_a ram=load ram=j
    tmp=select_bus_a tmp=bus_write
    addr=fetch jump=true;
cp8_mj_mi:
    // TMP <= RAM[J]
    tmp=select_bus_a tmp=register_load
    ram=select_bus_a ram=bus_write ram=j;
    // RAM[I] <= TMP, MPC <= #fetch
    ram=select_bus_a ram=load ram=i
    tmp=select_bus_a tmp=bus_write
    addr=fetch jump=true;
push_i:
    // RAM[--SP] <= I[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    i=select_bus_a i=rank_8 i=select_byte_2 i=bus_write;
    // RAM[--SP] <= I[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    i=select_bus_a i=rank_8 i=select_byte_1 i=bus_write
    addr=fetch jump=true;
push_j:
    // RAM[--SP] <= J[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    j=select_bus_a j=rank_8 j=select_byte_2 j=bus_write;
```

```
    // RAM[--SP] <= J[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    j=select_bus_a j=rank_8 j=select_byte_1 j=bus_write
    addr=fetch jump=true;
push_a:
    // RAM[--SP] <= A[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    a=select_bus_a a=rank_8 a=select_byte_2 a=bus_write;
    // RAM[--SP] <= A[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    a=select_bus_a a=rank_8 a=select_byte_1 a=bus_write
    addr=fetch jump=true;
push_b:
    // RAM[--SP] <= B[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    b=select_bus_a b=rank_8 b=select_byte_2 b=bus_write;
    // RAM[--SP] <= B[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    b=select_bus_a b=rank_8 b=select_byte_1 b=bus_write
    addr=fetch jump=true;
push_bp:
    // RAM[--SP] <= BP[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    bp=select_bus_a bp=rank_8 bp=select_byte_2 bp=bus_write;
    // RAM[--SP] <= BP[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    bp=select_bus_a bp=rank_8 bp=select_byte_1 bp=bus_write
    addr=fetch jump=true;
push_sp:
    // RAM[--SP] <= SP[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    sp=select_bus_a sp=rank_8 sp=select_byte_2 sp=bus_write;
    // RAM[--SP] <= SP[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    sp=select_bus_a sp=rank_8 sp=select_byte_1 sp=bus_write
    addr=fetch jump=true;
push_c:
    // RAM[--SP] <= C[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write;
    // RAM[--SP] <= C[7:0], MPC <= #fetch
```

```
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write
    addr=fetch jump=true;
push_fl:
    // RAM[--SP] <= FL[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    fl=select_bus_a fl=rank_8 fl=select_byte_2 fl=bus_write;
    // RAM[--SP] <= FL[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    fl=select_bus_a fl=rank_8 fl=select_byte_1 fl=bus_write
    addr=fetch jump=true;
push_num:
    // TMP[7:0] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // TMP[15:8] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // RAM[--SP] <= TMP[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_2 tmp=bus_write;
    // RAM[--SP] <= TMP[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_1 tmp=bus_write
    addr=fetch jump=true;
push_mi:
    // TMP[7:0] <= RAM[I]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=i;
    // TMP[15:8] <= RAM[I+1]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=i i=qp1;
    // RAM[--SP] <= TMP[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_2 tmp=bus_write;
    // RAM[--SP] <= TMP[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_1 tmp=bus_write
    addr=fetch jump=true;
push_mj:
    // TMP[7:0] <= RAM[J]
```

```
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=j;
    // TMP[15:8] <= RAM[J+1]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=j j=qp1;
    // RAM[--SP] <= TMP[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_2 tmp=bus_write;
    // RAM[--SP] <= TMP[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_1 tmp=bus_write
    addr=fetch jump=true;
pop_i:
    // I[7:0] <= RAM[SP++]
    i=select_bus_a i=rank_8 i=register_load_byte_1 i=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // I[15:8] <= RAM[SP++], MPC <= #fetch
    i=select_bus_a i=rank_8 i=register_load_byte_2 i=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop_j:
    // J[7:0] <= RAM[SP++]
    j=select_bus_a j=rank_8 j=register_load_byte_1 j=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // J[15:8] <= RAM[SP++], MPC <= #fetch
    j=select_bus_a j=rank_8 j=register_load_byte_2 j=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop_a:
    // A[7:0] <= RAM[SP++]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // A[15:8] <= RAM[SP++], MPC <= #fetch
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop_b:
    // B[7:0] <= RAM[SP++]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // B[15:8] <= RAM[SP++], MPC <= #fetch
```

```
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop_bp:
    // BP[7:0] <= RAM[SP++]
    bp=select_bus_a bp=rank_8 bp=register_load_byte_1 bp=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // BP[15:8] <= RAM[SP++], MPC <= #fetch
    bp=select_bus_a bp=rank_8 bp=register_load_byte_2 bp=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop_sp:
    // SP[7:0] <= RAM[SP++]
    sp=select_bus_a sp=rank_8 sp=register_load_byte_1 sp=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // SP[15:8] <= RAM[SP++], MPC <= #fetch
    sp=select_bus_a sp=rank_8 sp=register_load_byte_2 sp=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop_c:
    // C[7:0] <= RAM[SP++]
    c=select_bus_a c=rank_8 c=register_load_byte_1 c=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // C[15:8] <= RAM[SP++], MPC <= #fetch
    c=select_bus_a c=rank_8 c=register_load_byte_2 c=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop_fl:
    // FL[7:0] <= RAM[SP++]
    fl=select_bus_a fl=rank_8 fl=register_load_byte_1 fl=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // FL[15:8] <= RAM[SP++], MPC <= #fetch
    fl=select_bus_a fl=rank_8 fl=register_load_byte_2 fl=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop_mi:
    // TMP[7:0] <= RAM[SP++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // TMP[15:8] <= RAM[SP++], MPC <= #fetch
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
```

```
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // RAM[I] <= TMP[7:0]
    ram=select_bus_a ram=load ram=i
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_1 tmp=bus_write;
    // RAM[I+1] <= TMP[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=i i=qp1
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_2 tmp=bus_write
    addr=fetch jump=true;
pop_mj:
    // TMP[7:0] <= RAM[SP++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // TMP[15:8] <= RAM[SP++], MPC <= #fetch
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // RAM[J] <= TMP[7:0]
    ram=select_bus_a ram=load ram=j
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_1 tmp=bus_write;
    // RAM[J+1] <= TMP[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=j j=qp1
    tmp=select_bus_a tmp=rank_8 tmp=select_byte_2 tmp=bus_write
    addr=fetch jump=true;
push8_i:
    // RAM[--SP] <= I, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    i=select_bus_a i=bus_write
    addr=fetch jump=true;
push8_j:
    // RAM[--SP] <= J, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    j=select_bus_a j=bus_write
    addr=fetch jump=true;
push8_a:
    // RAM[--SP] <= A, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    a=select_bus_a a=bus_write
    addr=fetch jump=true;
push8_b:
    // RAM[--SP] <= B, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    b=select_bus_a b=bus_write
```

```
    addr=fetch jump=true;
push8_bp:
    // RAM[--SP] <= BP, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    bp=select_bus_a bp=bus_write
    addr=fetch jump=true;
push8_sp:
    // RAM[--SP] <= SP, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    sp=select_bus_a sp=bus_write
    addr=fetch jump=true;
push8_c:
    // RAM[--SP] <= C, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
push8_fl:
    // RAM[--SP] <= FL, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    fl=select_bus_a fl=bus_write
    addr=fetch jump=true;
push8_num:
    // TMP <= RAM[PC++]
    tmp=select_bus_a tmp=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // RAM[--SP] <= TMP, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    tmp=select_bus_a tmp=bus_write
    addr=fetch jump=true;
push8_mi:
    // TMP <= RAM[I]
    tmp=select_bus_a tmp=register_load
    ram=select_bus_a ram=bus_write ram=i;
    // RAM[--SP] <= TMP, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    tmp=select_bus_a tmp=bus_write
    addr=fetch jump=true;
push8_mj:
    // TMP <= RAM[J]
    tmp=select_bus_a tmp=register_load
    ram=select_bus_a ram=bus_write ram=j;
```

```
    // RAM[--SP] <= TMP, MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    tmp=select_bus_a tmp=bus_write
    addr=fetch jump=true;
pop8_i:
    // I <= RAM[SP++], MPC <= #fetch
    i=select_bus_a i=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop8_j:
    // J <= RAM[SP++], MPC <= #fetch
    j=select_bus_a j=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop8_a:
    // A <= RAM[SP++], MPC <= #fetch
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop8_b:
    // B <= RAM[SP++], MPC <= #fetch
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop8_bp:
    // BP <= RAM[SP++], MPC <= #fetch
    bp=select_bus_a bp=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop8_sp:
    // SP <= RAM[SP++], MPC <= #fetch
    sp=select_bus_a sp=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop8_c:
    // C <= RAM[SP++], MPC <= #fetch
    c=select_bus_a c=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop8_fl:
    // FL <= RAM[SP++], MPC <= #fetch
```

```
    fl=select_bus_a fl=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
pop8_mi:
    // TMP <= RAM[SP++]
    tmp=select_bus_a tmp=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // RAM[I] <= TMP, MPC <= #fetch
    ram=select_bus_a ram=load ram=i
    tmp=select_bus_a tmp=bus_write
    addr=fetch jump=true;
pop8_mj:
    // TMP <= RAM[SP++]
    tmp=select_bus_a tmp=register_load
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // RAM[J] <= TMP, MPC <= #fetch
    ram=select_bus_a ram=load ram=j
    tmp=select_bus_a tmp=bus_write
    addr=fetch jump=true;
jump:
    // TMP[7:0] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // TMP[15:8] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // PC <= TMP, MPC <= #fetch
    pc=select_bus_a pc=rank_16 pc=register_load
    tmp=select_bus_a tmp=rank_16 tmp=bus_write
    addr=fetch jump=true;
jump_nc:
    addr=jump jump=carry_f;
    // condition failed: ignore argument and jump to next instruction
    pc=qp2 pc=qup addr=fetch jump=true;
jump_c:
    addr=jump jump=carry_t;
    // condition failed: ignore argument and jump to next instruction
    pc=qp2 pc=qup addr=fetch jump=true;
jump_nz:
    addr=jump jump=zero_f;
    // condition failed: ignore argument and jump to next instruction
```

```
    pc=qp2 pc=qup addr=fetch jump=true;
jump_z:
    addr=jump jump=zero_t;
    // condition failed: ignore argument and jump to next instruction
    pc=qp2 pc=qup addr=fetch jump=true;
jump_nn:
    addr=jump jump=negative_f;
    // condition failed: ignore argument and jump to next instruction
    pc=qp2 pc=qup addr=fetch jump=true;
jump_n:
    addr=jump jump=negative_t;
    // condition failed: ignore argument and jump to next instruction
    pc=qp2 pc=qup addr=fetch jump=true;
jump_no:
    addr=jump jump=overflow_f;
    // condition failed: ignore argument and jump to next instruction
    pc=qp2 pc=qup addr=fetch jump=true;
jump_o:
    addr=jump jump=overflow_t;
    // condition failed: ignore argument and jump to next instruction
    pc=qp2 pc=qup addr=fetch jump=true;
//
equal:
    // C <= A (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_16
    addr=fetch jump=true;
not:
    // C <= ~A (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=not alu=rank_16
    addr=fetch jump=true;
and:
    // C <= A&B (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
```

```
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=and alu=rank_16
    addr=fetch jump=true;
nand:
    // C <= ~(A&B) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=nand alu=rank_16
    addr=fetch jump=true;
or:
    // C <= A|B (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=or alu=rank_16
    addr=fetch jump=true;
nor:
    // C <= ~(A|B) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=nor alu=rank_16
    addr=fetch jump=true;
xor:
    // C <= A^B (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=xor alu=rank_16
    addr=fetch jump=true;
nxor:
    // C <= ~(A^B) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
```

```
    alu=nxor alu=rank_16
    addr=fetch jump=true;
add:
    // C <= A+B (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=add alu=rank_16
    addr=fetch jump=true;
sub:
    // C <= A-B (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=sub alu=rank_16
    addr=fetch jump=true;
addc:
    // C <= A+B+carry (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=addc alu=rank_16
    addr=fetch jump=true;
subb:
    // C <= A-B-borrow (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=subb alu=rank_16
    addr=fetch jump=true;
lshl:
    // C <= lshl(A) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=lshl alu=rank_16
    addr=fetch jump=true;
```

```
lshr:
    // C <= lshr(A) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=lshr alu=rank_16
    addr=fetch jump=true;
ashl:
    // C <= ashl(A) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=ashl alu=rank_16
    addr=fetch jump=true;
ashr:
    // C <= ashr(A) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=ashr alu=rank_16
    addr=fetch jump=true;
rotcl:
    // C <= rotcl(A) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotcl alu=rank_16
    addr=fetch jump=true;
rotcr:
    // C <= rotcr(A) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotcr alu=rank_16
    addr=fetch jump=true;
rotl:
    // C <= rotl(A) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotl alu=rank_16
```

```
    addr=fetch jump=true;
rotr:
    // C <= rotr(A) (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotr alu=rank_16
    addr=fetch jump=true;
add8:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=add alu=rank_8
    addr=fetch jump=true;
sub8:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=sub alu=rank_8
    addr=fetch jump=true;
add8c:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=addc alu=rank_8
    addr=fetch jump=true;
sub8b:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=subb alu=rank_8
    addr=fetch jump=true;
lsh8l:
```

```
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=lshl alu=rank_8
    addr=fetch jump=true;
lsh8r:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=lshr alu=rank_8
    addr=fetch jump=true;
ash8l:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=ashl alu=rank_8
    addr=fetch jump=true;
ash8r:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=ashr alu=rank_8
    addr=fetch jump=true;
rot8cl:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotcl alu=rank_8
    addr=fetch jump=true;
rot8cr:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotcr alu=rank_8
    addr=fetch jump=true;
```

```
rot8l:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotl alu=rank_8
    addr=fetch jump=true;
rot8r:
    // C <= A[7:0]+B[7:0] (FL updated), MPC <= #fetch
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotr alu=rank_8
    addr=fetch jump=true;

cast_uns_a:
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_8;
    c=select_bus_a c=bus_write
    a=select_bus_a a=register_load
    addr=fetch jump=true;
cast_uns_b:
    b=select_bus_a b=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_8;
    c=select_bus_a c=bus_write
    b=select_bus_a b=register_load
    addr=fetch jump=true;
cast_sig_a:
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load c=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_8 alu=signed;
    c=select_bus_a c=bus_write c=rank_16
    a=select_bus_a a=register_load a=rank_16
    addr=fetch jump=true;
cast_sig_b:
    b=select_bus_a b=bus_write
```

```
    c=select_bus_b c=register_load c=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_8 alu=signed;
    c=select_bus_a c=bus_write c=rank_16
    b=select_bus_a b=register_load b=rank_16
    addr=fetch jump=true;
equals:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= A (FL updated), MPC <= #fetch
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_16
    addr=fetch jump=true;
nots:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= ~A (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=not alu=rank_16;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
ands:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
```

```
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= A&B (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=and alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
nands:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= ~(A&B) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
```

```
    alu=nand alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
ors:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= A|B (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=or alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
nors:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
```

```
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= ~(A|B) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=nor alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
xors:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= A^B (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=xor alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
```

```
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
nxors:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= ~(A^B) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=nxor alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
adds:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
```

```
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= A+B (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=add alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
subs:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= A-B (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=sub alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
```

```
addcs:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= A+B+carry (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=addc alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
subbs:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // C <= A-B-borrow (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
```

```
    b=select_bus_b b=rank_16 b=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=subb alu=rank_16;
    // RAM[SP=SP+2] <= C[7:0]
    ram=select_bus_a ram=load ram=s sp=qp2 sp=qup
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
lshls:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= lshl(A) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=lshl alu=rank_16;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
lshrs:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= lshr(A) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
```

```
    alu=lshr alu=rank_16;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
ashls:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= ashl(A) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=ashl alu=rank_16;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
ashrs:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= ashr(A) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=ashr alu=rank_16;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
```

```
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
rotcls:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= rotcl(A) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotcl alu=rank_16;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
rotcrs:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= rotcr(A) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotcr alu=rank_16;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
```

```
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
rotls:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= rotl(A) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotl alu=rank_16;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
rotrs:
    // A[7:0] <= RAM[SP]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // A[15:8] <= RAM[SP+1]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= rotr(A) (FL updated)
    a=select_bus_a a=rank_16 a=bus_write
    c=select_bus_b c=rank_16 c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotr alu=rank_16;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
add8s:
```

```
    // B <= RAM[SP]
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // A <= RAM[SP+1]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= A+B (FL updated)
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=add alu=rank_8;
    // RAM[SP=SP+1] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1 sp=qup
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
sub8s:
    // B <= RAM[SP]
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // A <= RAM[SP+1]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= A-B (FL updated)
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=sub alu=rank_8;
    // RAM[SP=SP+1] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1 sp=qup
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
add8cs:
    // B <= RAM[SP]
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // A <= RAM[SP+1]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= A+B+carry (FL updated)
```

```
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=addc alu=rank_8;
    // RAM[SP=SP+1] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1 sp=qup
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
sub8bs:
    // B <= RAM[SP]
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // A <= RAM[SP+1]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // C <= A-B-borrow (FL updated)
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=subb alu=rank_8;
    // RAM[SP=SP+1] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1 sp=qup
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
lsh8ls:
    // A <= RAM[SP]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // C <= lsh8l(A) (FL updated)
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=lshl alu=rank_8;
    // RAM[SP] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
lsh8rs:
    // A <= RAM[SP]
```

```
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // C <= lsh8r(A) (FL updated)
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=lshr alu=rank_8;
    // RAM[SP] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
ash8ls:
    // A <= RAM[SP]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // C <= ash8l(A) (FL updated)
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=ashl alu=rank_8;
    // RAM[SP] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
ash8rs:
    // A <= RAM[SP]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // C <= ash8r(A) (FL updated)
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=ashr alu=rank_8;
    // RAM[SP] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
rot8cls:
    // A <= RAM[SP]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
```

```
    // C <= rot8cl(A) (FL updated)
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotcl alu=rank_8;
    // RAM[SP] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
rot8crs:
    // A <= RAM[SP]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // C <= rot8cr(A) (FL updated)
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotcr alu=rank_8;
    // RAM[SP] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
rot8ls:
    // A <= RAM[SP]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // C <= rot8l(A) (FL updated)
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotl alu=rank_8;
    // RAM[SP] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
rot8rs:
    // A <= RAM[SP]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // C <= rot8r(A) (FL updated)
    a=select_bus_a a=bus_write
```

```
    c=select_bus_b c=register_load
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=rotr alu=rank_8;
    // RAM[SP] <= C[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=bus_write
    addr=fetch jump=true;
casts_uns:
    // A <= RAM[SP]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // C <= A[7:0] (FL updated)
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load c=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_8;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
casts_sig:
    // A <= RAM[SP]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // C <= A[7:0] (FL updated)
    a=select_bus_a a=bus_write
    c=select_bus_b c=register_load c=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_8 alu=signed;
    // RAM[SP] <= C[7:0]
    ram=select_bus_a ram=load ram=s
    c=select_bus_a c=rank_8 c=select_byte_1 c=bus_write;
    // RAM[SP+1] <= C[15:8], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qp1
    c=select_bus_a c=rank_8 c=select_byte_2 c=bus_write
    addr=fetch jump=true;
call:
    // TMP[7:0] <= RAM[PC++]
```

```
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // TMP[15:8] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // RAM[--SP] <= PC[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_2 pc=bus_write;
    // RAM[--SP] <= PC[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_1 pc=bus_write;
    // PC <= TMP, MPC <= #fetch
    pc=select_bus_a pc=rank_16 pc=register_load
    tmp=select_bus_a tmp=rank_16 tmp=bus_write
    addr=fetch jump=true;
call_i:
    // RAM[--SP] <= PC[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_2 pc=bus_write;
    // RAM[--SP] <= PC[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_1 pc=bus_write;
    // PC <= I, MPC <= #fetch
    pc=select_bus_a pc=rank_16 pc=register_load
    i=select_bus_a i=rank_16 i=bus_write
    addr=fetch jump=true;
call_j:
    // RAM[--SP] <= PC[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_2 pc=bus_write;
    // RAM[--SP] <= PC[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_1 pc=bus_write;
    // PC <= J, MPC <= #fetch
    pc=select_bus_a pc=rank_16 pc=register_load
    j=select_bus_a j=rank_16 j=bus_write
    addr=fetch jump=true;
return:
    // PC[7:0] <= RAM[SP++]
    pc=select_bus_a pc=rank_8 pc=register_load_byte_1 pc=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
```

```
    // PC[15:8] <= RAM[SP++], MPC <= #fetch
    pc=select_bus_a pc=rank_8 pc=register_load_byte_2 pc=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
ivtl:
    // TMP[7:0] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_1 tmp=select_byte_1
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // TMP[15:8] <= RAM[PC++]
    tmp=select_bus_a tmp=rank_8 tmp=register_load_byte_2 tmp=select_byte_2
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // IVT <= TMP, MPC <= #fetch
    ivt=ivt_load
    tmp=select_bus_a tmp=rank_16 tmp=bus_write
    addr=fetch jump=true;
int:
    //
    // push FL
    //
    // RAM[--SP] <= FL[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    fl=select_bus_a fl=rank_8 fl=select_byte_2 fl=bus_write;
    // RAM[--SP] <= FL[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    fl=select_bus_a fl=rank_8 fl=select_byte_1 fl=bus_write;
    //
    // Reset interrupt enable flag
    //
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=cleari;
    //
    // push (PC+1) to jump over the interrupt argument.
    //
    // RAM[--SP] <= (PC+1)[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_2 pc=bus_write pc=qp1;
    // RAM[--SP] <= (PC+1)[7:0]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_1 pc=bus_write pc=qp1;
    //
    // Convert the interrupt number from the argument, to the
```

```
    // interrupt vector table item address:
    //     TMP <= IVT[int], int <= RAM[PC]
    //
    // TMP <= IVT[RAM[PC]]
    tmp=select_bus_a tmp=rank_16 tmp=register_load
    ivt=select_int_b ivt=bus_write
    ram=select_bus_b ram=bus_write ram=p;
    //
    // Prepare to call the interrupt routine, loading the
    // interrupt vector table item address into PC.
    //
    // PC[7:0] <= RAM[TMP]
    pc=select_bus_a pc=rank_8 pc=register_load_byte_1 pc=select_byte_1
    ram=select_bus_a ram=bus_write ram=t;
    // PC[15:8] <= RAM[TMP+1], MPC <= #fetch
    pc=select_bus_a pc=rank_8 pc=register_load_byte_2 pc=select_byte_2
    ram=select_bus_a ram=bus_write ram=t tmp=qp1
    addr=fetch jump=true;
iret:
    //
    // Pop PC
    //
    // PC[7:0] <= RAM[SP++]
    pc=select_bus_a pc=rank_8 pc=register_load_byte_1 pc=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // PC[15:8] <= RAM[SP++]
    pc=select_bus_a pc=rank_8 pc=register_load_byte_2 pc=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    //
    // Pop FL
    //
    // FL[7:0] <= RAM[SP++]
    fl=select_bus_a fl=rank_8 fl=register_load_byte_1 fl=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=dp1;
    // FL[15:8] <= RAM[SP++], MPC <= #fetch
    fl=select_bus_a fl=rank_8 fl=register_load_byte_2 fl=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=dp1
    addr=fetch jump=true;
imrl:
    // IRQ <= RAM[PC++], MPC <= #fetch
    irq=irq_mask_load
```

```
    ram=select_bus_a ram=bus_write ram=p pc=dp1
    addr=fetch jump=true;
irq:
    //
    // push FL
    //
    // RAM[--SP] <= FL[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    fl=select_bus_a fl=rank_8 fl=select_byte_2 fl=bus_write;
    // RAM[--SP] <= FL[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    fl=select_bus_a fl=rank_8 fl=select_byte_1 fl=bus_write;
    //
    // Reset interrupt enable flag
    //
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=cleari;
    //
    // push (PC-1) to jump back to the opcode just read.
    //
    // RAM[--SP] <= (PC-1)[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_2 pc=bus_write pc=qm1;
    // RAM[--SP] <= (PC-1)[7:0]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_1 pc=bus_write pc=qm1;
    //
    // Convert the interrupt number from the IRQ module, to the
    // interrupt vector table item address:
    //     TMP <= IVT[int], int <= IRQ
    //
    // TMP <= IVT[INT[IRQ]]
    tmp=select_bus_a tmp=rank_16 tmp=register_load
    ivt=select_int_a ivt=bus_write;
    //
    // Prepare to call the interrupt routine, loading the
    // interrupt vector table item address into PC.
    //
    // PC[7:0] <= RAM[TMP]
    pc=select_bus_a pc=rank_8 pc=register_load_byte_1 pc=select_byte_1
    ram=select_bus_a ram=bus_write ram=t;
```

```
    // PC[15:8] <= RAM[TMP+1]
    pc=select_bus_a pc=rank_8 pc=register_load_byte_2 pc=select_byte_2
    ram=select_bus_a ram=bus_write ram=t tmp=qp1;
    //
    // Set the IRQ received as already done,
    // MPC <= #fetch
    irq=irq_done
    addr=fetch jump=true;
op_error:
    //
    // push FL
    //
    // RAM[--SP] <= FL[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    fl=select_bus_a fl=rank_8 fl=select_byte_2 fl=bus_write;
    // RAM[--SP] <= FL[7:0], MPC <= #fetch
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    fl=select_bus_a fl=rank_8 fl=select_byte_1 fl=bus_write;
    //
    // Reset interrupt enable flag
    //
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=cleari;
    //
    // push (PC-1) to jump back to the opcode just read.
    //
    // RAM[--SP] <= (PC-1)[15:8]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_2 pc=bus_write pc=qm1;
    // RAM[--SP] <= (PC-1)[7:0]
    ram=select_bus_a ram=load ram=s sp=qm1 sp=qup
    pc=select_bus_a pc=rank_8 pc=select_byte_1 pc=bus_write pc=qm1;
    //
    // Convert the interrupt number 0 to the
    // interrupt vector table item address:
    //
    // TMP <= IVT[0]
    tmp=select_bus_a tmp=rank_16 tmp=register_load
    ivt=select_int_b ivt=bus_write
    bus=bw bus=select_bus_b bus=0;
    //
```

```
    // Prepare to call the interrupt routine, loading the
    // interrupt vector table item address into PC.
    //
    // PC[7:0] <= RAM[TMP]
    pc=select_bus_a pc=rank_8 pc=register_load_byte_1 pc=select_byte_1
    ram=select_bus_a ram=bus_write ram=t;
    // PC[15:8] <= RAM[TMP+1], MPC <= #fetch
    pc=select_bus_a pc=rank_8 pc=register_load_byte_2 pc=select_byte_2
    ram=select_bus_a ram=bus_write ram=t tmp=qp1
    addr=fetch jump=true;
flag_i_clr:
    // FL <= FL&0xFFEF, MPC <= #fetch
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=cleari irq=irq_done
    addr=fetch jump=true;
flag_i_set:
    // FL <= FL|0x0010, MPC <= #fetch
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=seti irq=irq_done
    addr=fetch jump=true;
flag_c_clr:
    // FL <= FL&0xFFFE, MPC <= #fetch
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=clearc
    addr=fetch jump=true;
flag_c_set:
    // FL <= FL&0x0001, MPC <= #fetch
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=setc
    addr=fetch jump=true;
in_num_a:
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // I/O request
    ioc=req;
    // Time to wait.
    ctrl=nop;
    // A <= IOC, MPC <= #fetch
    a=select_bus_a a=register_load ioc=bus_write
    addr=fetch jump=true;
```

```
in_num_b:
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // I/O request
    ioc=req;
    // Time to wait.
    ctrl=nop;
    // B <= IOC, MPC <= #fetch
    b=select_bus_a b=register_load ioc=bus_write
    addr=fetch jump=true;
in_num_mi:
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // I/O request
    ioc=req;
    // Time to wait.
    ctrl=nop;
    // RAM[I] <= IOC, MPC <= #fetch
    ram=select_bus_a ram=load ram=i ioc=bus_write
    addr=fetch jump=true;
in_num_mj:
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // I/O request
    ioc=req;
    // Time to wait.
    ctrl=nop;
    // RAM[J] <= IOC, MPC <= #fetch
    ram=select_bus_a ram=load ram=j ioc=bus_write
    addr=fetch jump=true;
out_a_num:
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // IOC <= A
    ioc=load a=select_bus_a a=bus_write;
    // I/O request, MPC <= #fetch
    ioc=req
```

```
    addr=fetch jump=true;
out_b_num:
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // IOC <= B
    ioc=load b=select_bus_a b=bus_write;
    // I/O request, MPC <= #fetch
    ioc=req
    addr=fetch jump=true;
out_mi_num:
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // IOC <= RAM[I]
    ioc=load ram=select_bus_a ram=bus_write ram=i;
    // I/O request, MPC <= #fetch
    ioc=req
    addr=fetch jump=true;
out_mj_num:
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // IOC <= RAM[J]
    ioc=load ram=select_bus_a ram=bus_write ram=j;
    // I/O request, MPC <= #fetch
    ioc=req
    addr=fetch jump=true;
ifack_jump:
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // C <= alu(I/O isack) (FL updated), if ack MPC <= #jump
    ioc=isack ioc=bus_write
    c=select_bus_b c=register_load c=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_8 alu=signed
    addr=jump jump=zero_f;
    // discard argument and load next opcode
    pc=qp2 pc=qup addr=fetch jump=true;
ifack_call:
```

```
    // IOA <= RAM[PC++]
    ioa=select_bus_a ioa=register_load
    ram=select_bus_a ram=bus_write ram=p pc=dp1;
    // C <= alu(I/O isack) (FL updated), if ack MPC <= #ifack_call_ok
    ioc=isack ioc=bus_write
    c=select_bus_b c=register_load c=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=a alu=rank_8 alu=signed
    addr=call jump=zero_f;
    // discard argument and load next opcode
    pc=qp2 pc=qup addr=fetch jump=true;
cmpr:
    // FL(A-B), MPC <= #fetch
    a=select_bus_a a=bus_write a=rank_16
    b=select_bus_b b=bus_write b=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=sub alu=rank_16
    addr=fetch jump=true;
testr:
    // FL(A&B), MPC <= #fetch
    a=select_bus_a a=bus_write a=rank_16
    b=select_bus_b b=bus_write b=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=and alu=rank_16
    addr=fetch jump=true;
cmpi:
    // A[7:0] <= RAM[I]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=i;
    // A[15:8] <= RAM[I+1], MPC <= #fetch
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=i i=qp1
    // B[7:0] <= RAM[J]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=j;
    // B[15:8] <= RAM[J+1], MPC <= #fetch
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=j j=qp1;
    // FL(A-B), MPC <= #fetch
    a=select_bus_a a=bus_write a=rank_16
    b=select_bus_b b=bus_write b=rank_16
```

```
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=sub alu=rank_16
    addr=fetch jump=true;
testi:
    // A[7:0] <= RAM[I]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=i;
    // A[15:8] <= RAM[I+1], MPC <= #fetch
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=i i=qp1
    // B[7:0] <= RAM[J]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=j;
    // B[15:8] <= RAM[J+1], MPC <= #fetch
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=j j=qp1;
    // FL(A&B), MPC <= #fetch
    a=select_bus_a a=bus_write a=rank_16
    b=select_bus_b b=bus_write b=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=and alu=rank_16
    addr=fetch jump=true;
cmps:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // FL(A-B), MPC <= #fetch
    a=select_bus_a a=bus_write a=rank_16
    b=select_bus_b b=bus_write b=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=sub alu=rank_16
    addr=fetch jump=true;
```

```
tests:
    // B[7:0] <= RAM[SP]
    b=select_bus_a b=rank_8 b=register_load_byte_1 b=select_byte_1
    ram=select_bus_a ram=bus_write ram=s;
    // B[15:8] <= RAM[SP+1]
    b=select_bus_a b=rank_8 b=register_load_byte_2 b=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // A[7:0] <= RAM[SP+2]
    a=select_bus_a a=rank_8 a=register_load_byte_1 a=select_byte_1
    ram=select_bus_a ram=bus_write ram=s sp=qp2;
    // A[15:8] <= RAM[SP+3]
    a=select_bus_a a=rank_8 a=register_load_byte_2 a=select_byte_2
    ram=select_bus_a ram=bus_write ram=s sp=qp3;
    // FL(A&B), MPC <= #fetch
    a=select_bus_a a=bus_write a=rank_16
    b=select_bus_b b=bus_write b=rank_16
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=and alu=rank_16
    addr=fetch jump=true;
cmp8i:
    // A <= RAM[I]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=i;
    // B <= RAM[J]
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=j;
    // FL(A-B), MPC <= #fetch
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=sub alu=rank_8
    addr=fetch jump=true;
test8i:
    // A <= RAM[I]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=i;
    // B <= RAM[J]
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=j;
    // FL(A&B), MPC <= #fetch
    a=select_bus_a a=bus_write
```

```
    b=select_bus_b b=bus_write
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=and alu=rank_8
    addr=fetch jump=true;
cmp8s:
    // B <= RAM[SP]
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // A <= RAM[SP+1]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // FL(A-B), MPC <= #fetch
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=sub alu=rank_8
    addr=fetch jump=true;
test8s:
    // B <= RAM[SP]
    b=select_bus_a b=register_load
    ram=select_bus_a ram=bus_write ram=s;
    // A <= RAM[SP+1]
    a=select_bus_a a=register_load
    ram=select_bus_a ram=bus_write ram=s sp=qp1;
    // FL(A&B), MPC <= #fetch
    a=select_bus_a a=bus_write
    b=select_bus_b b=bus_write
    fl=select_bus_b fl=rank_16 fl=register_load
    alu=and alu=rank_8
    addr=fetch jump=true;
stop:
    ctrl=stop;
    addr=fetch jump=true;
//
end
```

11.16 Macrocode

«

As an example, the following listing shows a simple program that reads from the keyboard and prints the same text on the virtual screen. The keyboard input is read after a hardware interrupt.

Listing 11.44. Macrocode example for TKGate.

```
begin macrocode @ 0
        jump #start
        nop
interrupt_vector_table:
        .short  0x0025  // CPU opcode error
        .short  0x0024  // CPU
        .short  0x0024  // CPU
        .short  0x0024  // CPU
        .short  0x0024  // IRQ rtc
        .short  0x0026  // IRQ keyboard
        .short  0x0024  // IRQ hard disk
        .short  0x0024  // IRQ
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
default_interrupt_routine:
        iret
op_code_error:
        stop
keyboard:
        in 1,%A                         // keyboard read
        equal                           // update flags
        jump %zero,1,#keyboard_end      // if zero exit
        out %A,0                        // print on screen
        jump #keyboard                  // continue
keyboard_end:
```

```
        iret
start:
        cp 0x0080, %I                       // set stack bottom
        cp %I, %SP
        //
        ivtl #interrupt_vector_table
        //
        imrl 0x0F                           // all IRQ accepted
        flag_i 1                            // IRQ enabled
//
keyboard_reset:
        in 1,%A                             // keyboard read.
        equal                               // update flags
        jump %zero,0,#keyboard_reset        // if not zero continue
//
loop:
        jump #loop
stop:                   // never reach the end.
        stop
end
```

12.1 General purpose modules 734
12.2 Module "MEM" 739
12.2.1 Module "MAR" 740
12.2.2 Module "MIR" 742
12.2.3 Module "MDR" 743
12.3 General purpose registers 744
12.4 Module "ALU" and related submodules 748
12.5 Module "BUS" 755
12.6 Module "IRQ" 756
12.7 Module "IVT" 757
12.8 Module "RTC": real time clock 758
12.9 Module "TTY" 759
12.10 Module "HDD" 763
12.11 Module "CTRL" 768
12.11.1 Modules "F*n*" 769
12.11.2 Module "CLK_CLR" 770
12.12 Memory and microcode fields 772
12.13 Opcodes 776
12.14 Microcode 806
12.15 Macrocode 845

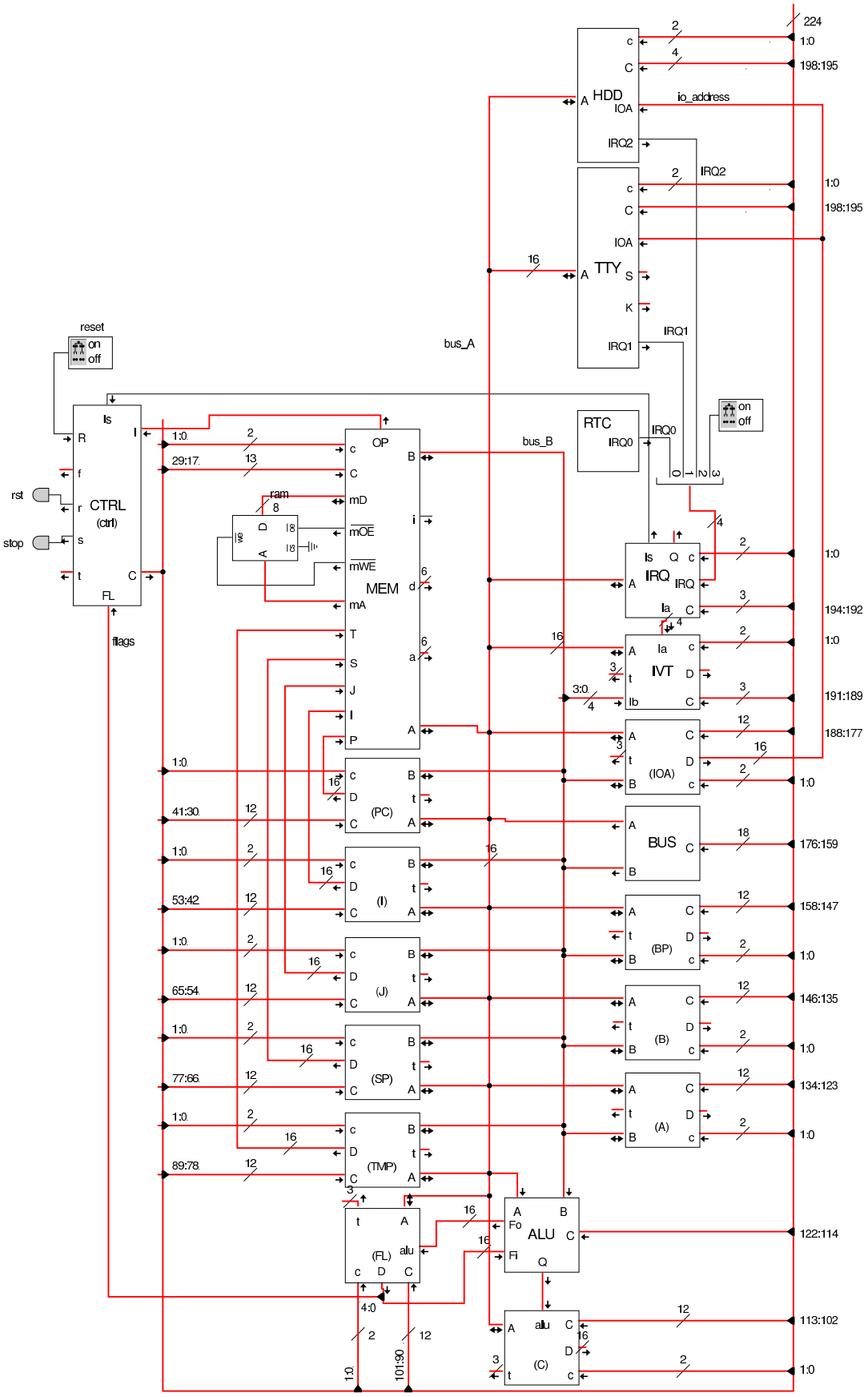
This version of the project is made to resolve the memory access problem previously evidenced. There is a module **MEM** that includes an ***MAR*** register for the address, the ***MIR*** register (*memory instruction register*, ex ***IR***) and a renewed ***MDR*** register. To

"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

access the memory, it is required to put the address inside the ***MAR*** register, specifying the rank and the direction (read or write); then the read or write starts and stops automatically. But this process requires three clock cycles for 16-bit rank and two clock cycles for 8-bit rank. As the register ***MAR*** is restored and it is necessarily traversed by the data read or written to the RAM memory, the rank adaptation functions are carried out by ***MDR*** and the other registers do not need that adaptation anymore.

Attachments	Description
attachments/xcpu/xcpu-e.v	TKGate Verilog netlist source file.
attachments/xcpu/xcpu-e.gm	TKGate microcode and macrocode source file.
attachments/xcpu/xcpu-e-terminal.vpd.tcl	TCL/Tk script related to the terminal module **`TTY`**.

Figure 12.2. Simple CPU, version “E”.



12.1 General purpose modules

The following modules were already used on the previous version of the project, except **vr1**, which is a counter able to increment or decrement only by 2^n values.

Figure 12.3. Rank reduction module: the module **RANK** and its submodules.

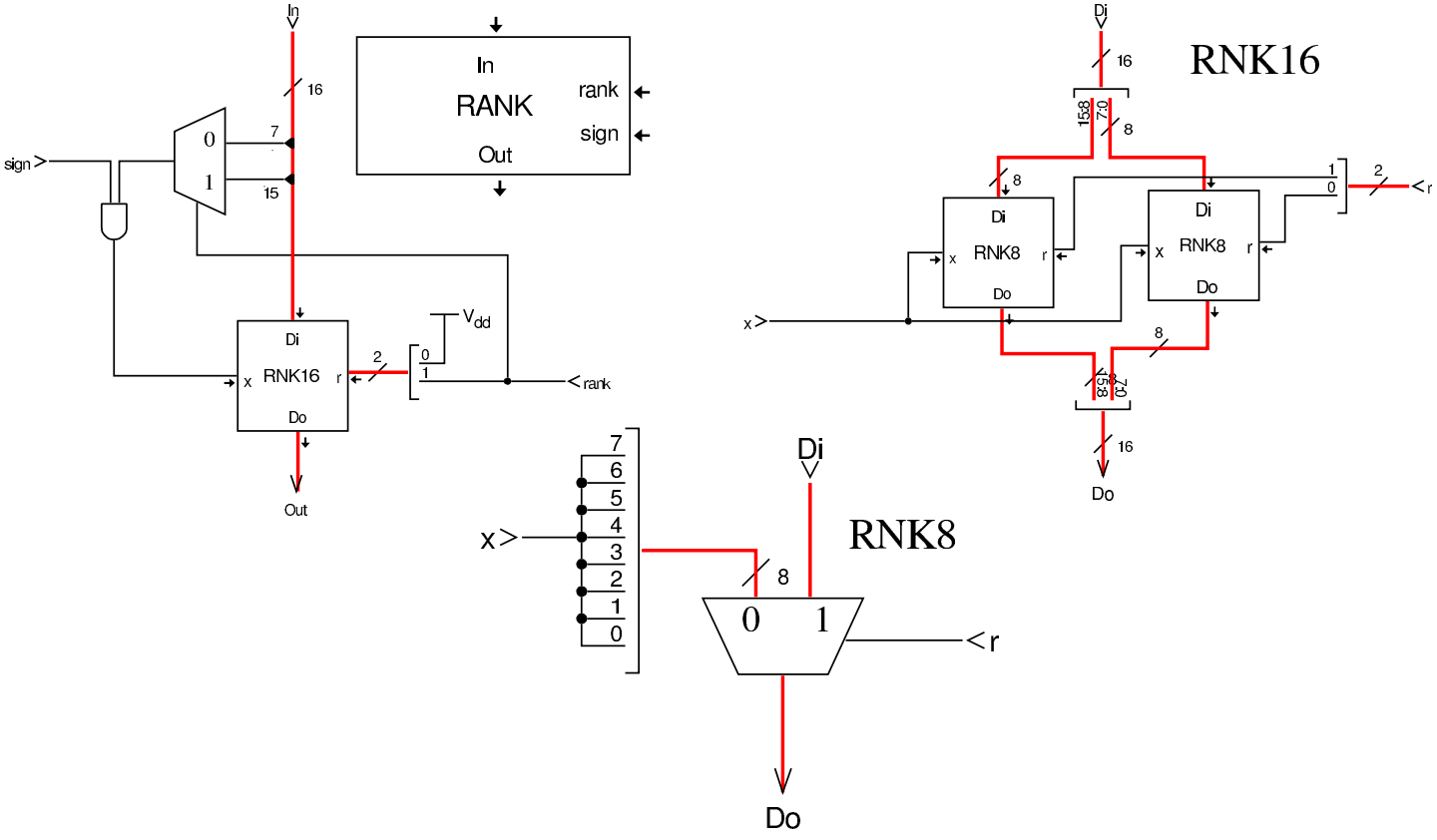


Figure 12.4. Building **D*n*** modules.

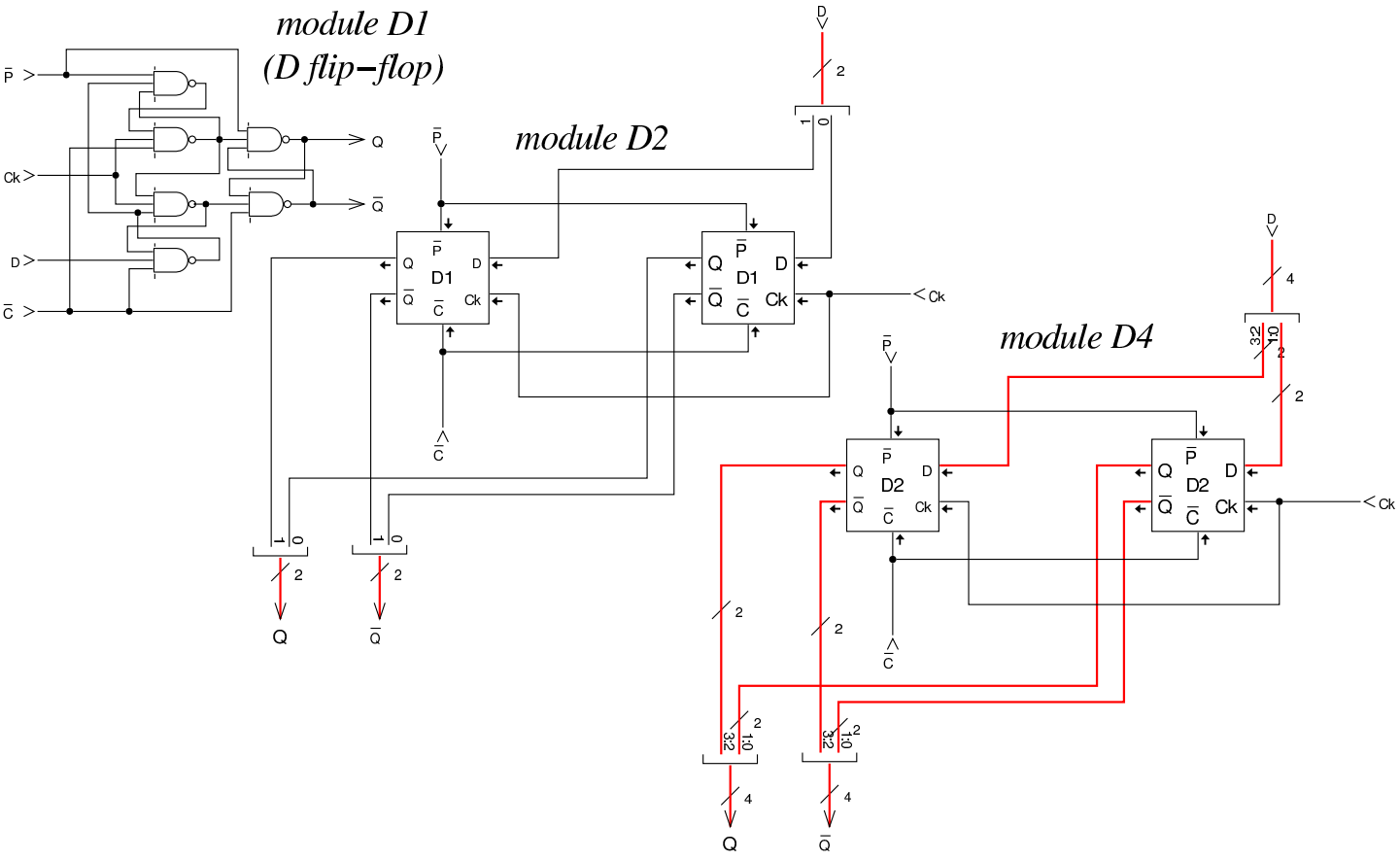


Figure 12.5. Building **DR*n*** modules.

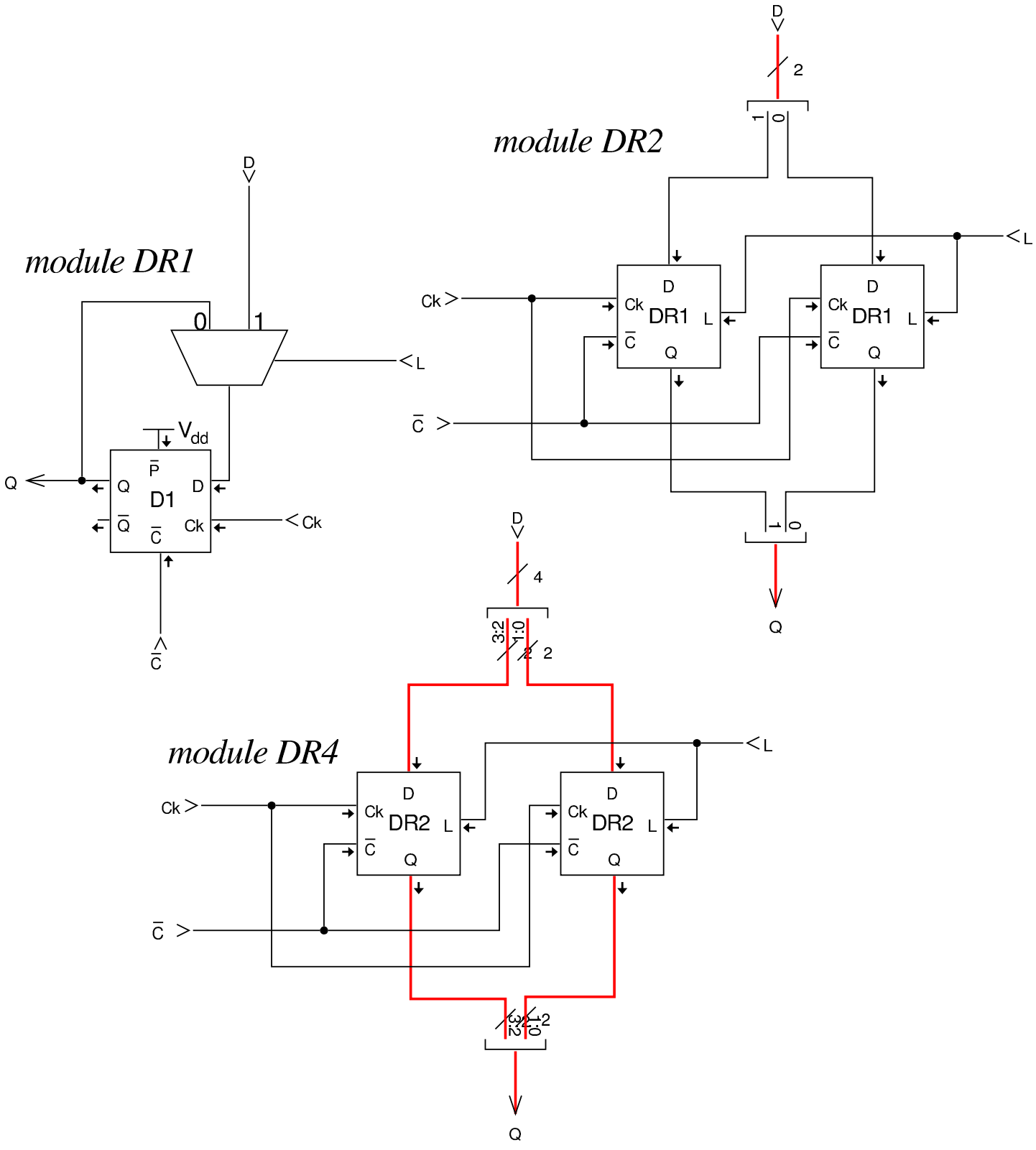


Figure 12.6. Building **DH*n*** modules.

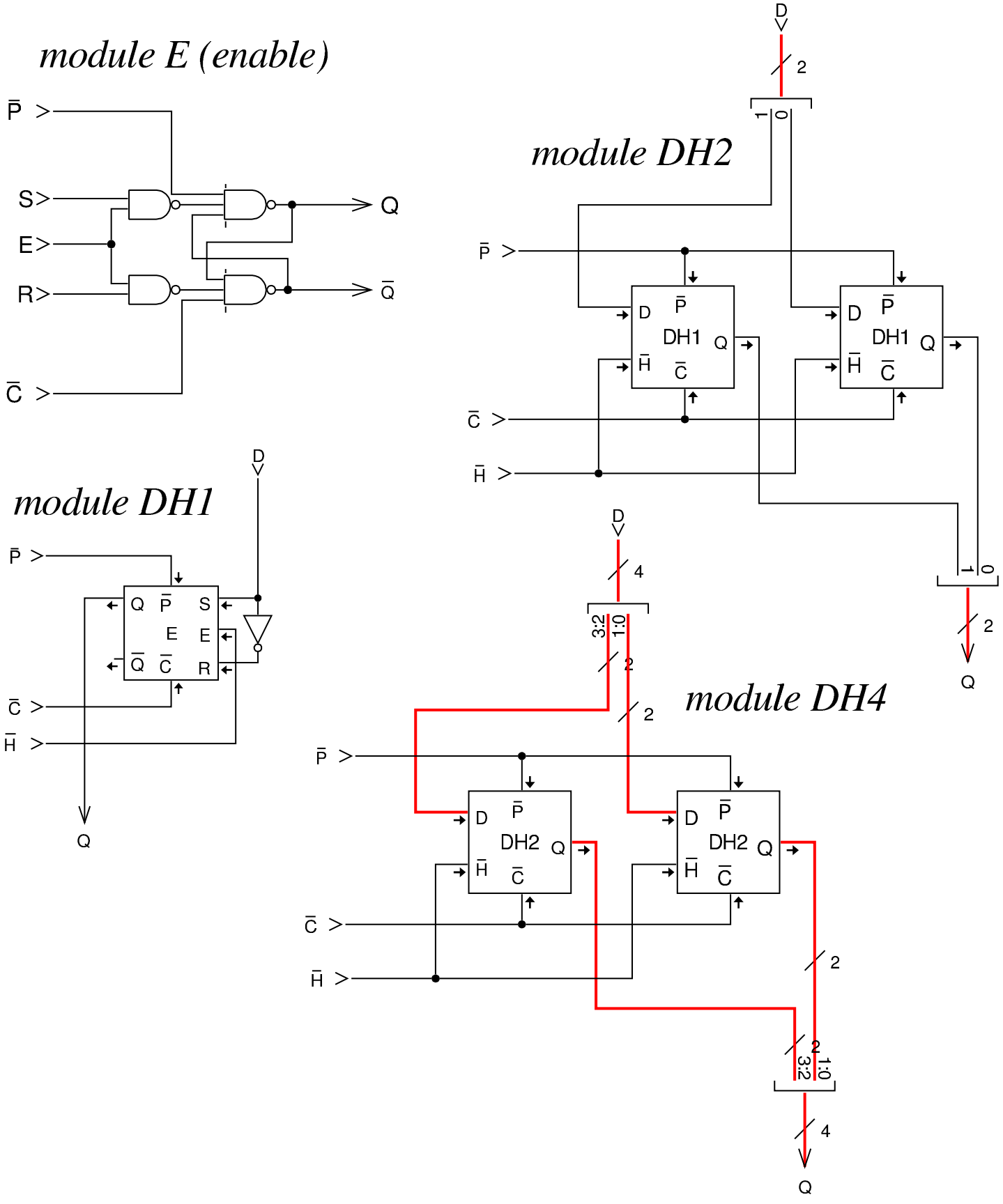


Figure 12.7. Building **vrn** modules.

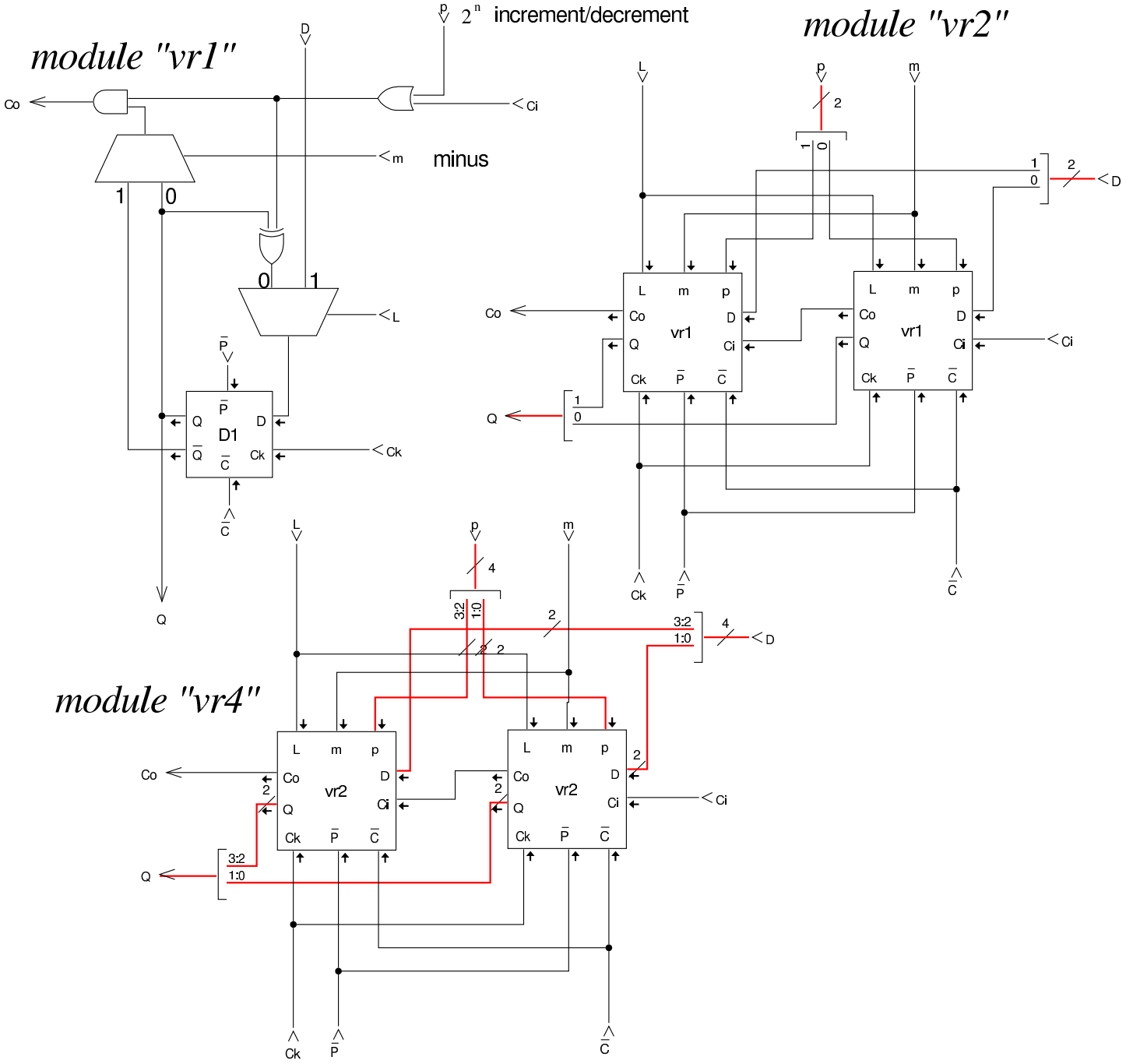
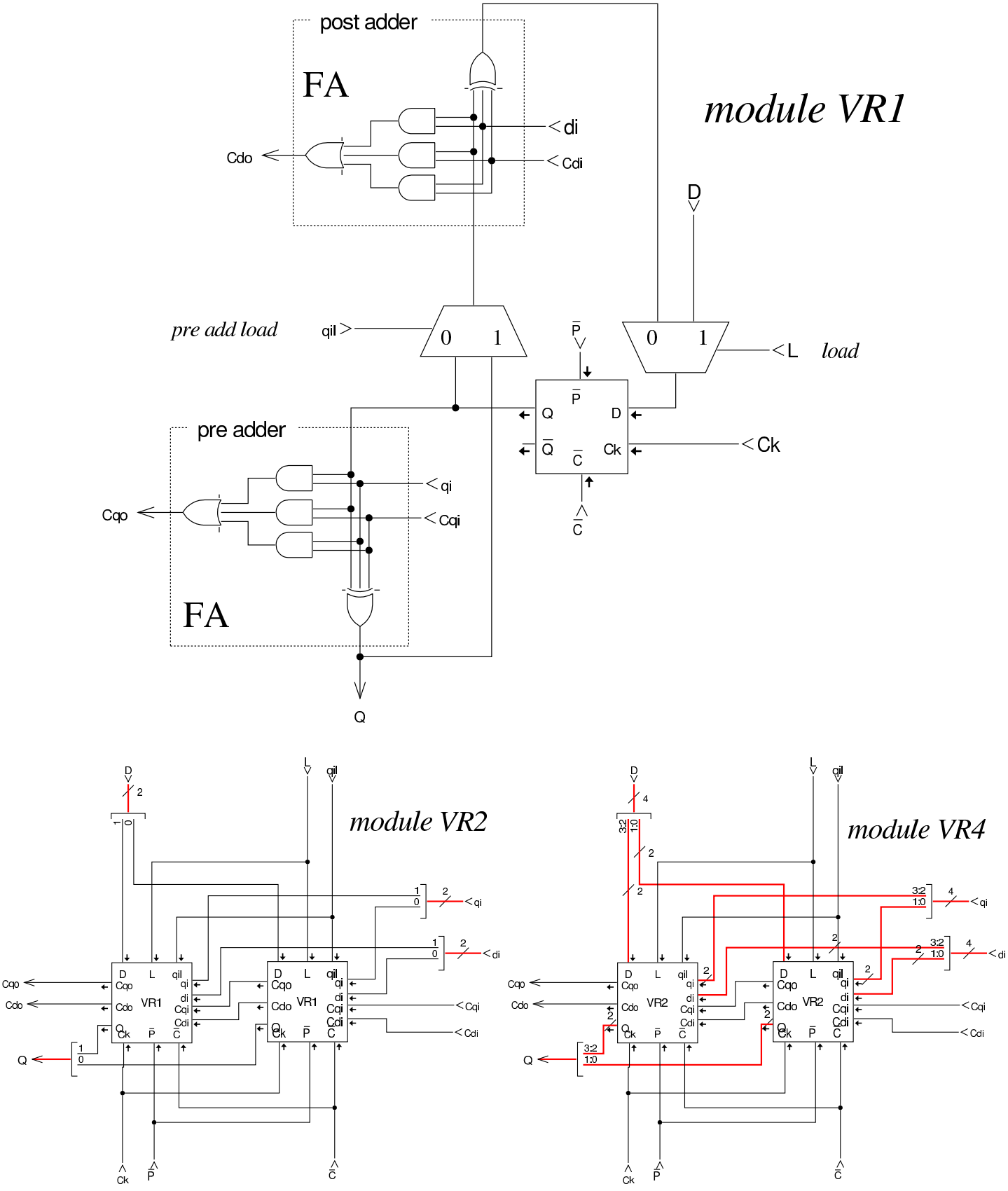


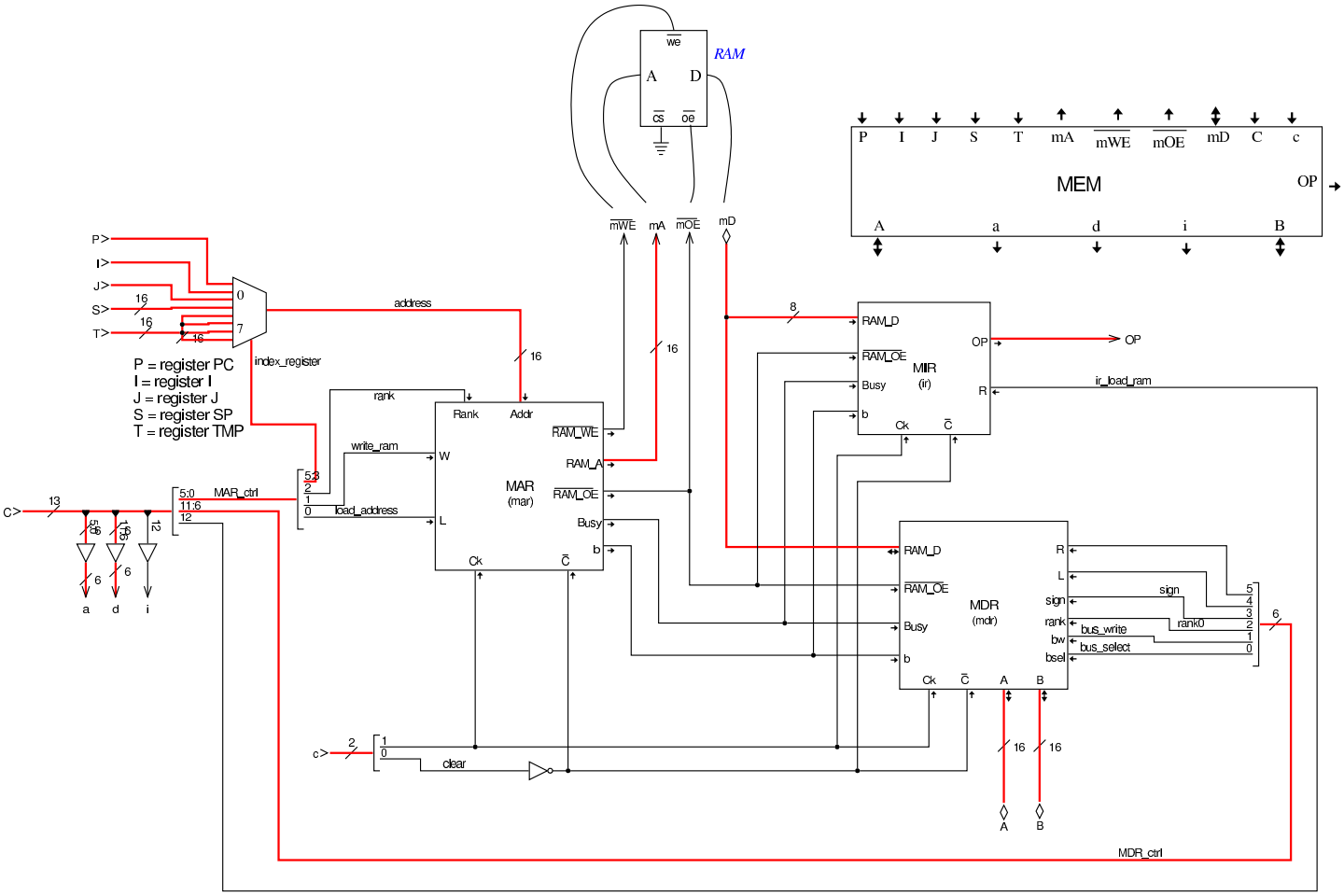
Figure 12.8. Building **VR*n*** modules.



12.2 Module "MEM"

The most important feature of this version is related to the memory management: the new module **MEM** contains the registers ***MAR***, ***MDR*** and ***MIR*** which were previously autonomous.

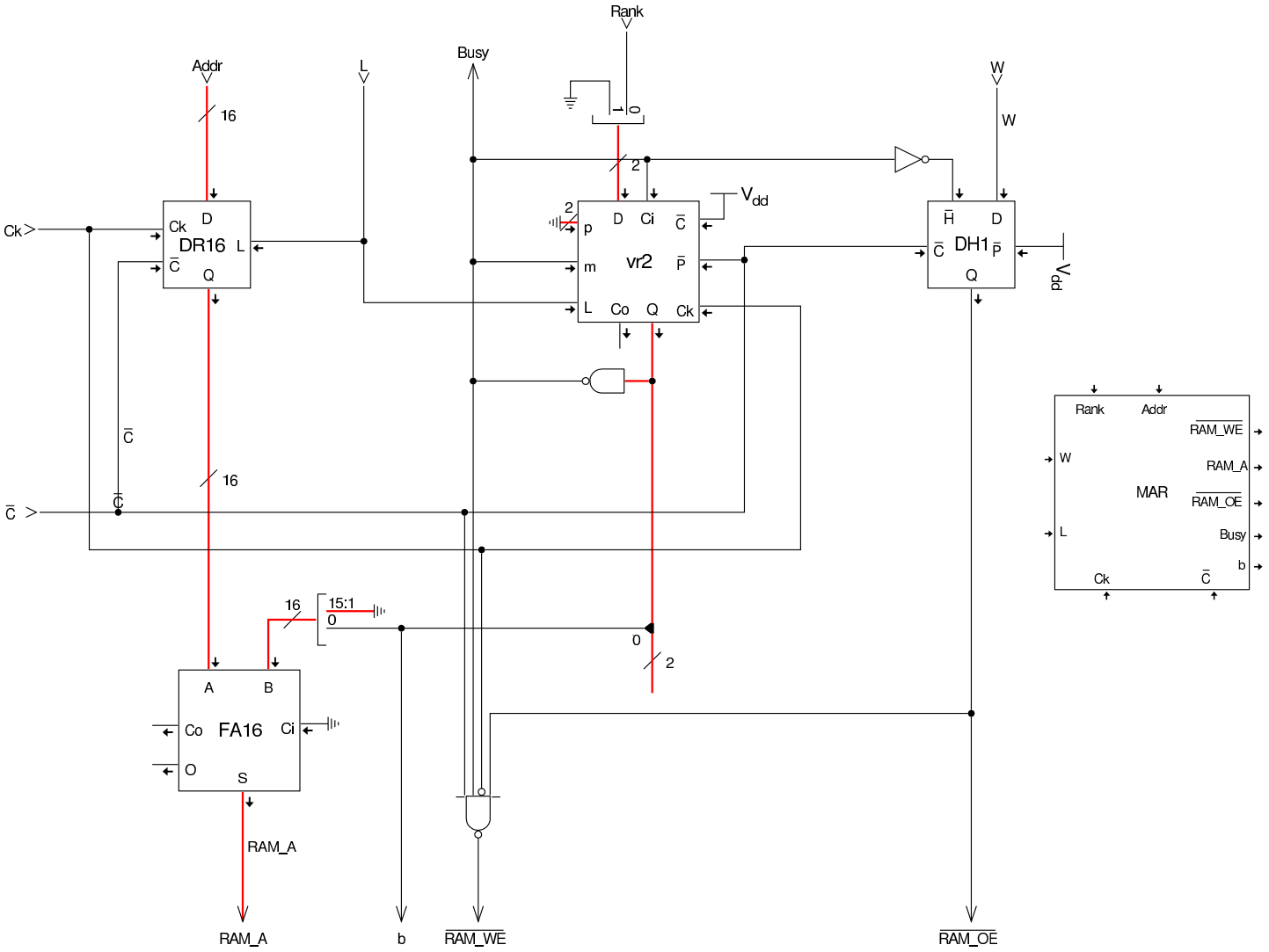
Figure 12.9. The module **MEM**.



12.2.1 Module "MAR"

To access the RAM it is necessary to specify the address of the first byte, the rank starting from that address and the direction (read or write). First it is the **MAR** module involved.

Figure 12.10. Module **MAR**.



Inside the above picture, should be noted the register used to store the memory address (on the left, type **DR16**). In the middle, the register of type **vr2** is responsible to hold and count the rank: zero means one byte; one means two bytes; -1 (11_2) means "stop". The middle register is not reset, but preset, so its initial condition is equal to the value -1.

When the rank counter is not equal to -1 (11_2), then the rank counter is automatically decremented at each clock; after the rank counter reach the zero, it becomes equal to -1 and it stops changing. That is: when the requested rank is one (meaning two byte), its first value is one, then zero, then -1. When the rank counter is loaded with a valid rank it becomes busy and the output ***Busy*** is used to transmit this notion to the other modules (**MDR** and **MIR**); at the same time, the latch on the right (type **DH1**) holds the value received from the input ***W***.

The output ***RAM_A*** is obtained as the sum of the original address received from input ***Addr*** and the rank value, reported by the rank counter. That is: the first address output by ***Addr*** is the highest. The output ***b*** is used to let know the other modules which byte of the selected rank is considered during a particular time. The output $\overline{\textbf{\textit{RAM\_OE}}}$ is asserted (zero as the meaning is complemented) when the RAM is to be written, the busy line is active and the clock signal is low.

The output $\overline{\textbf{\textit{RAM\_OE}}}$ is asserted (zero as the meaning is complemented) when the RAM is to be read: the output is negated (one) only when a write request is received.

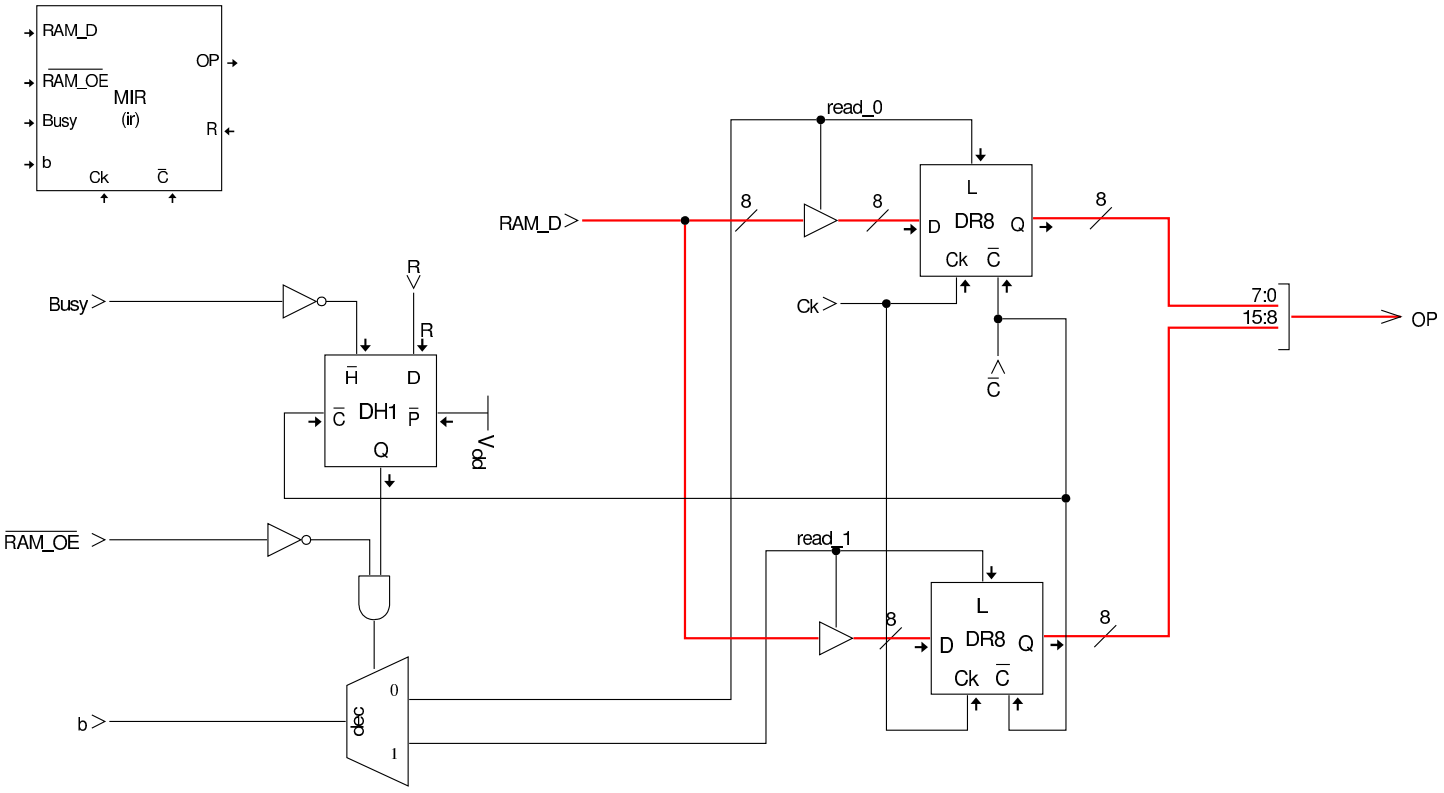
The RAM memory allows to be read in any condition, but when it comes the time to write it, it is necessary to activate the $\overline{\textbf{\textit{WE}}}$ input only when the address is already available: that is why it is allowed to write only when the clock is low, as no change on the address can happen.

12.2.2 Module "MIR"

«

The **MIR** module (*memory instruction register*) is responsible for reading from memory the next opcode to be executed. The current opcode occupy only one byte, but the **MIR** is prepared to accept a longer size: the ***b*** (byte) input line selects the byte that is currently read (the rank is specified to the **MAR** module). The $\overline{\textbf{\textit{RAM\_OE}}}$ input line is asserted when the RAM is read and the ***Busy*** line is active when the RAM access cycle is not yet finished. The input line ***R*** should be active when the **MIR** module should read from RAM; then, when the read cycle starts, the ***Busy*** becomes active and the read request is hold by the **DH1** module, up to the cycle end.

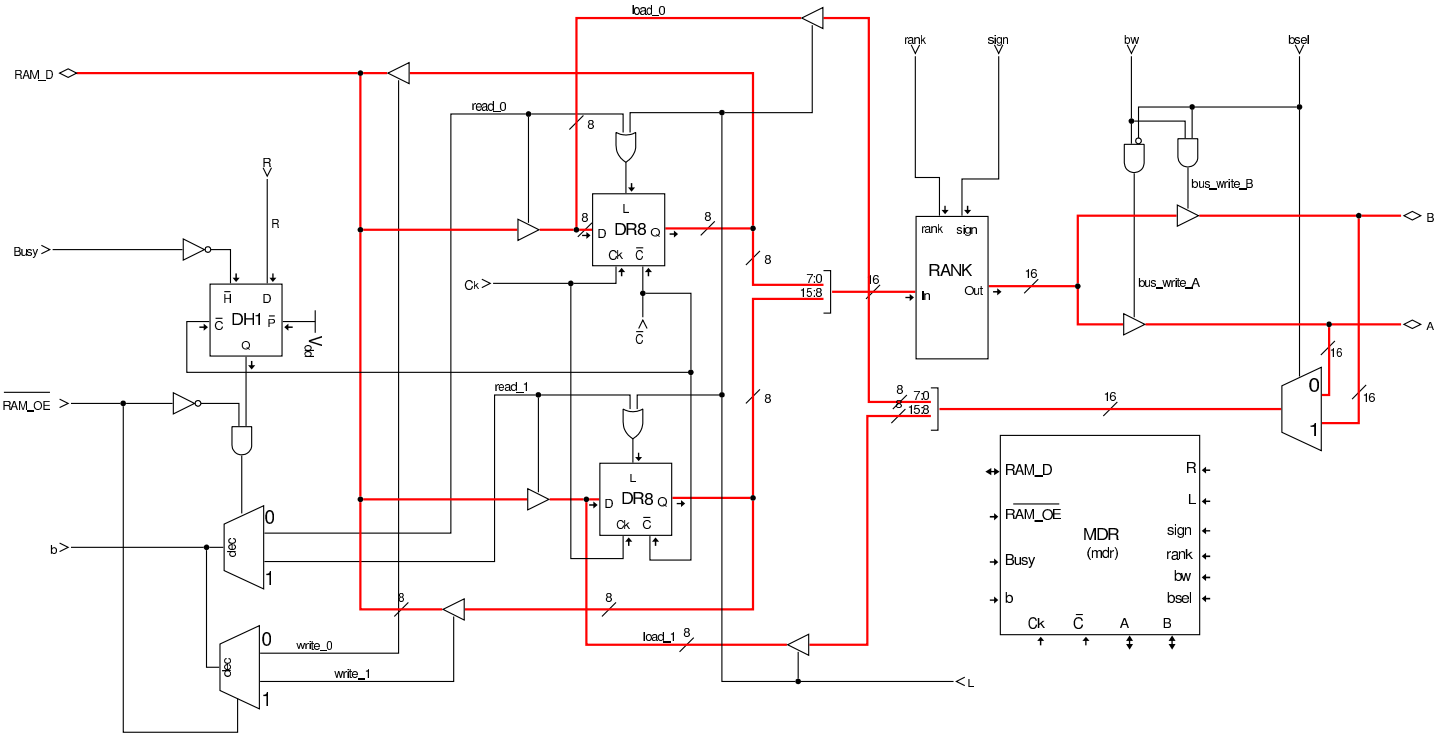
Figure 12.11. Module **MIR**.



12.2.3 Module "MDR"

The **MDR** module is a register used as buffer between the RAM memory and the CPU data bus. The difference from the **MIR** modules depends on the fact that it is connected to the two data busses and that can also write to the RAM. The **MDR** module is also responsible for rank adaptation of data, before writing to the data bus.

Figure 12.12. Module **MDR**.



12.3 General purpose registers

The registers are made in a simpler way compared to the previous version, because there is no rank problem: all data is moved at a 16-bit rank. There are two types of registers: the following figure shows the two external layouts.

Figure 12.13. 16-bit registers: external appearance.

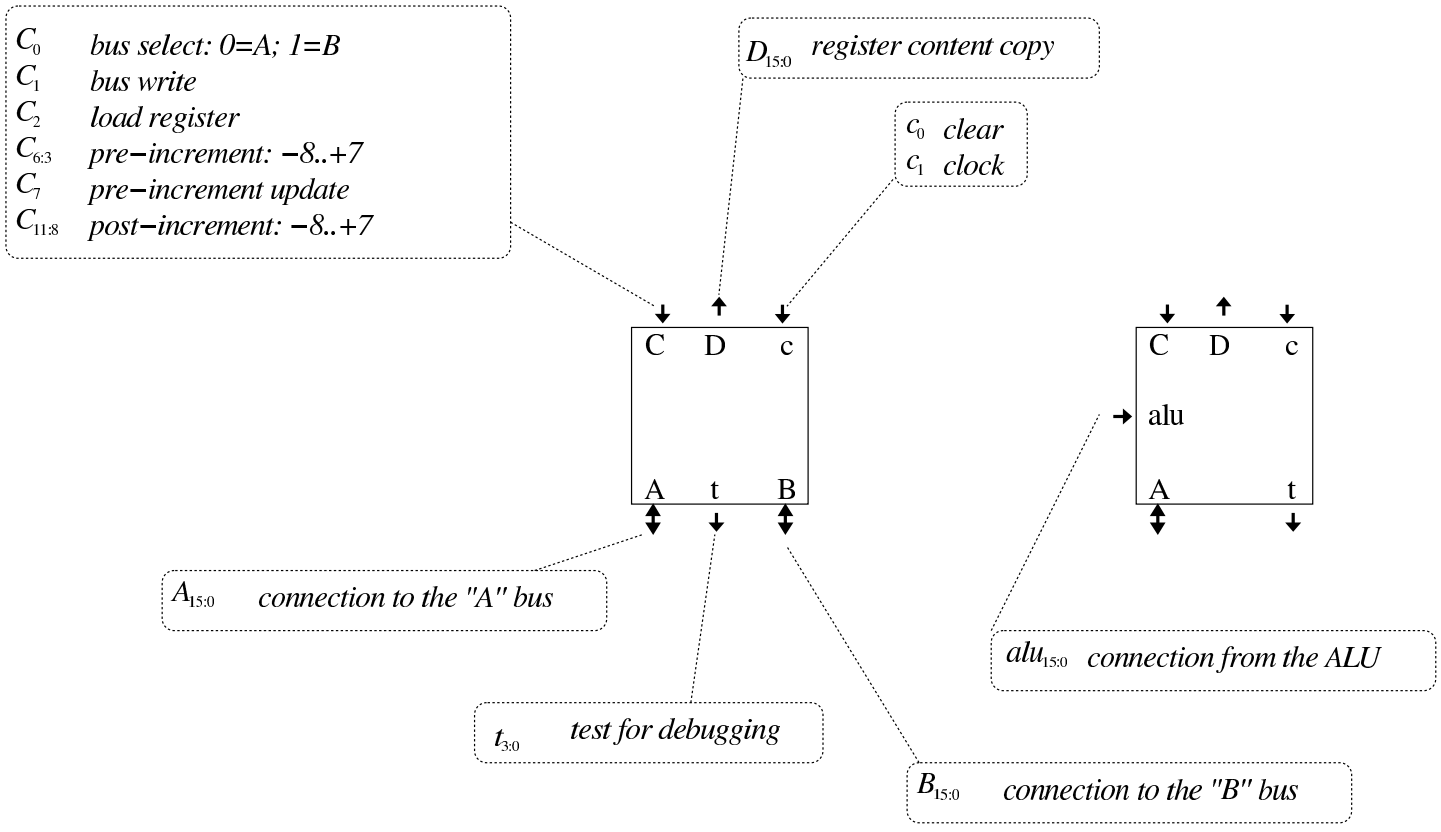
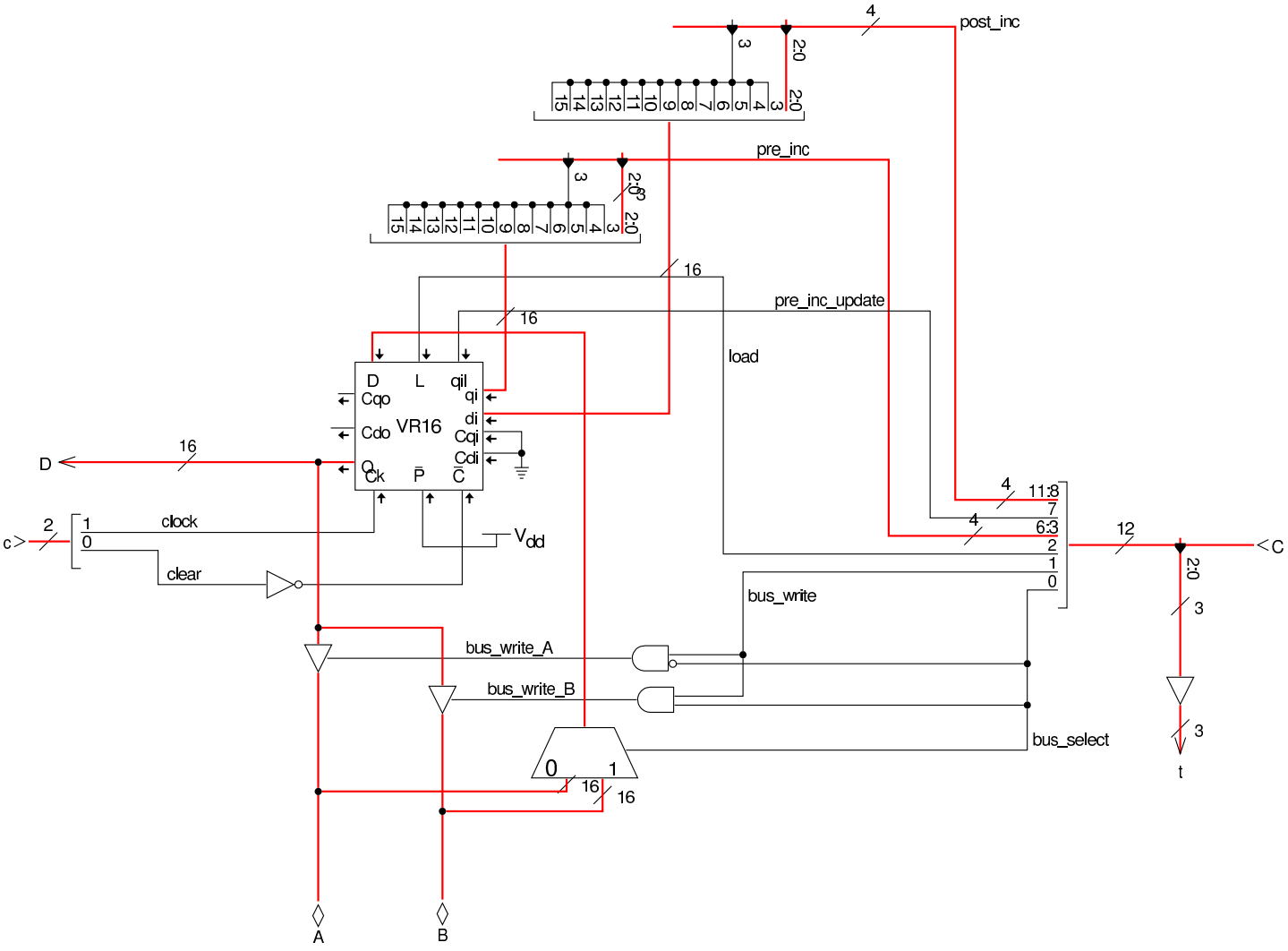
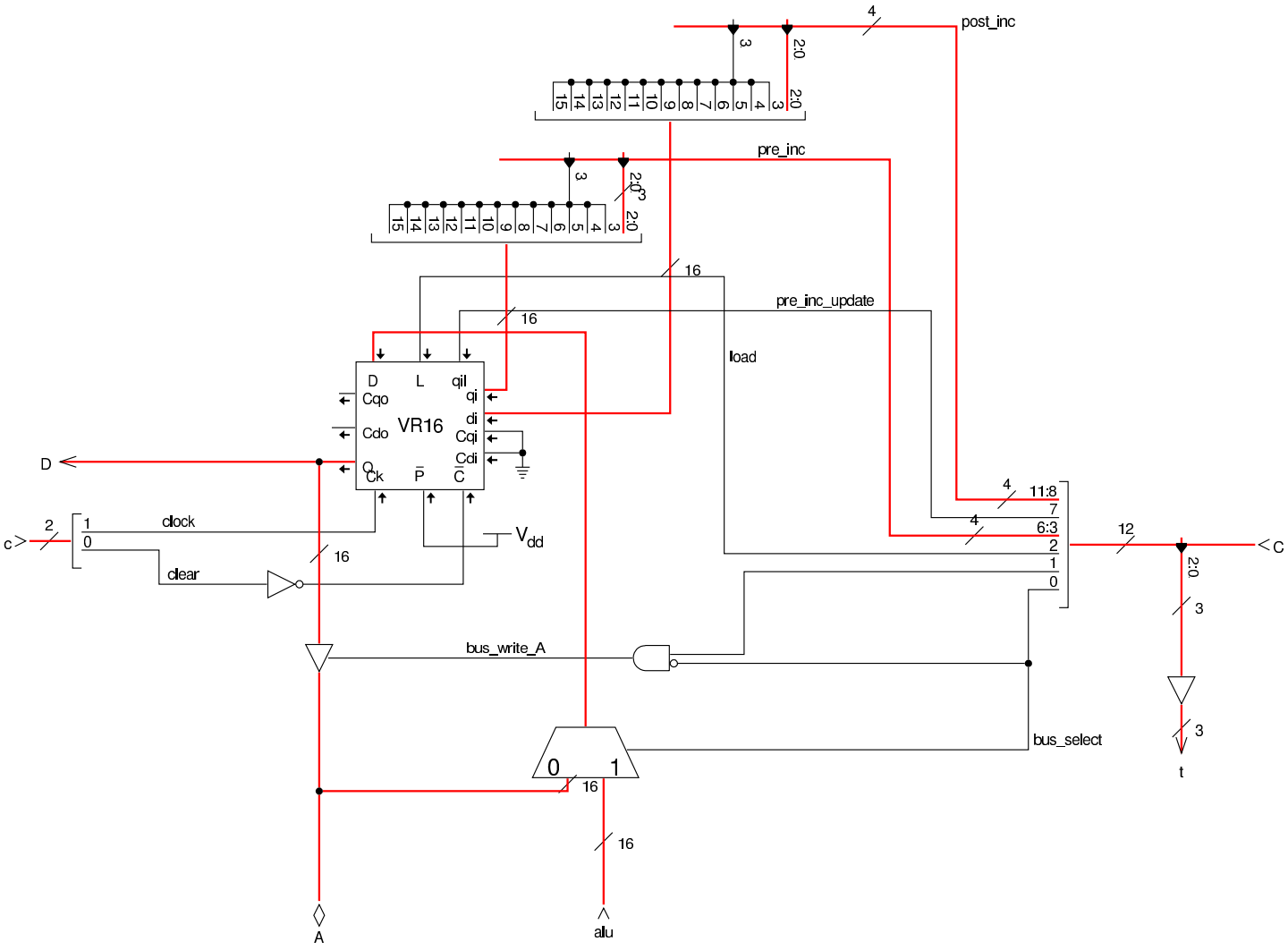


Figure 12.14. 16-bit registers: internal structure.



The ***C*** and ***FL*** registers are made with a slightly different module, where the "B" bus connection is named ***alu*** and is only able to receive data.

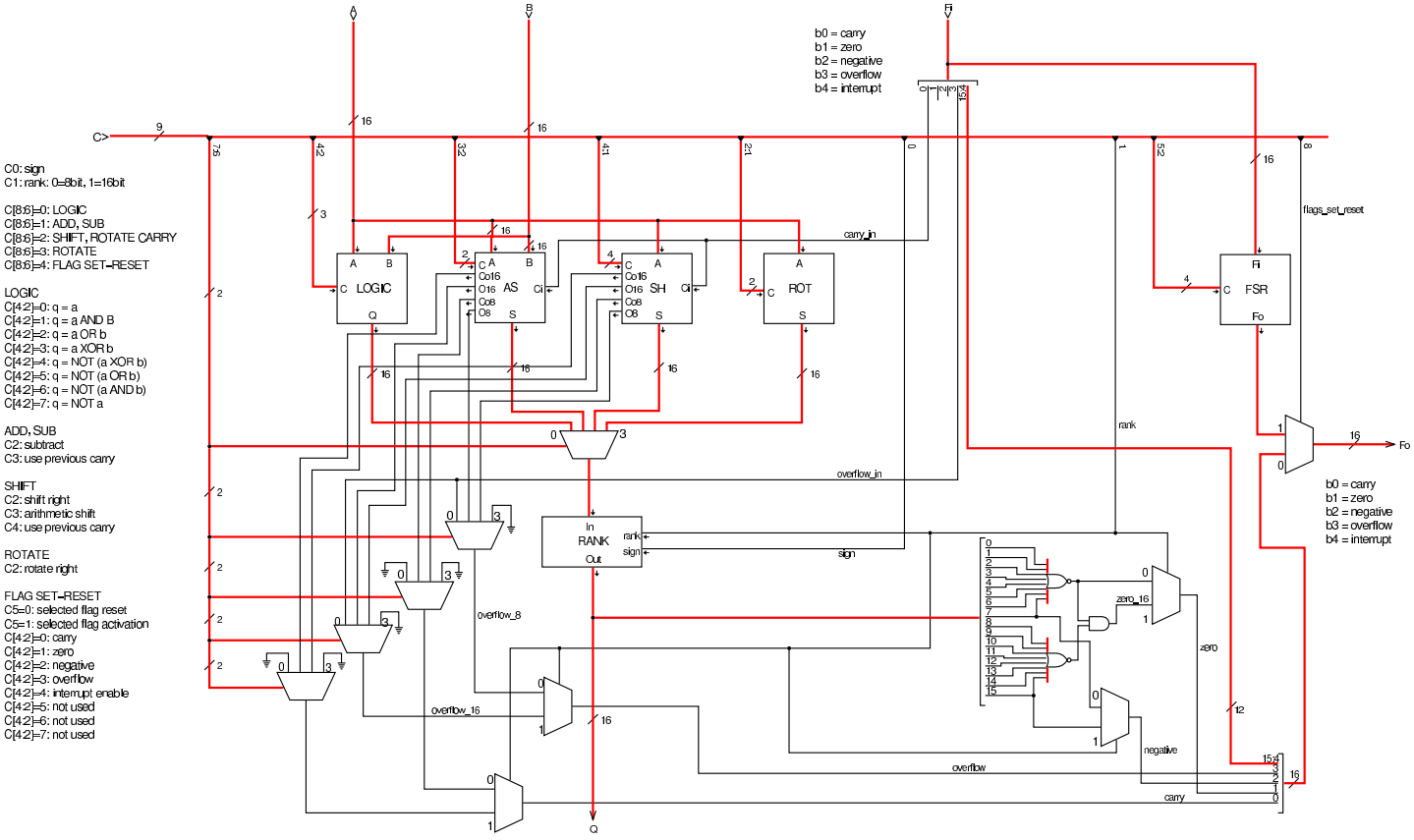
Figure 12.15. Alternative 16-bit registers structure.



12.4 Module "ALU" and related submodules

The module **ALU**, is the same as the previous version.

Figure 12.16. Module **ALU**.



The other components inside the **ALU** module are just the same as the previous version.

Figure 12.17. Module **LOGIC**: the logic unit.

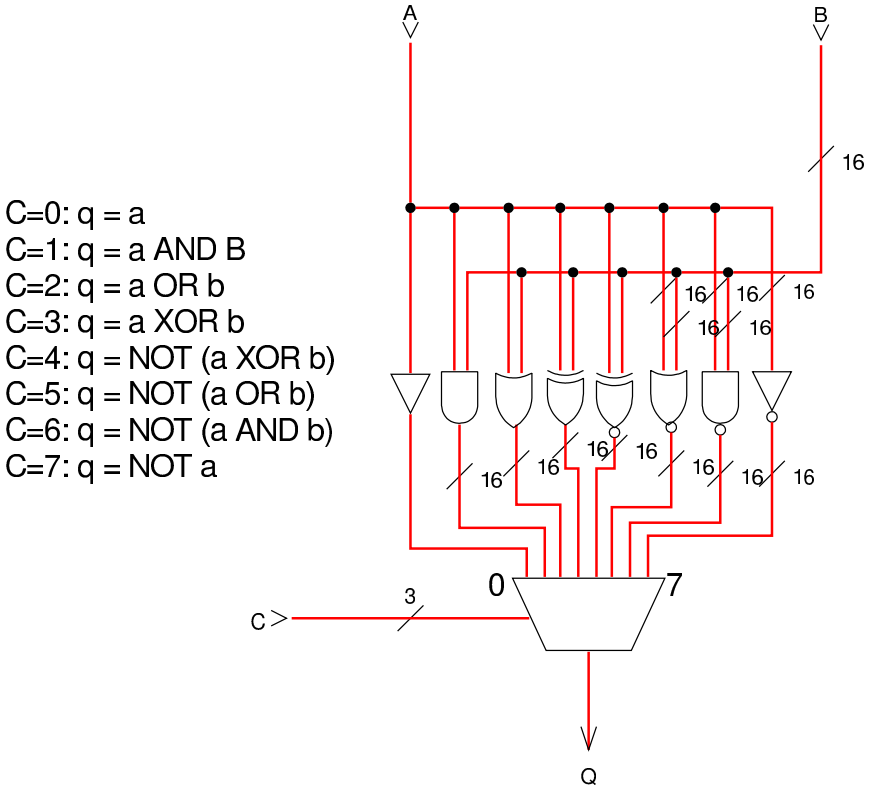


Figure 12.18. Module **AS**: addition and subtraction

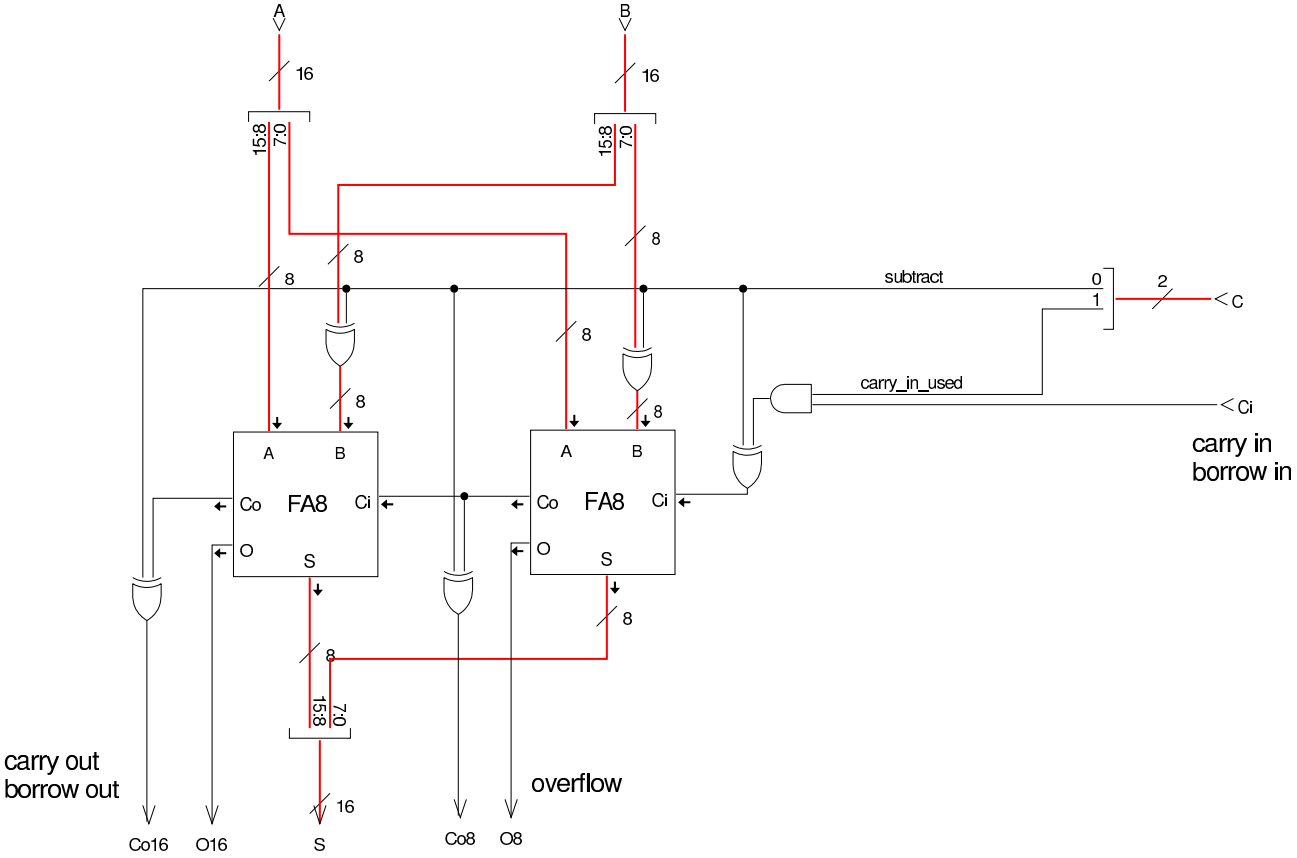


Figure 12.19. Modules **FA*n***: ***n***-bit full adder.

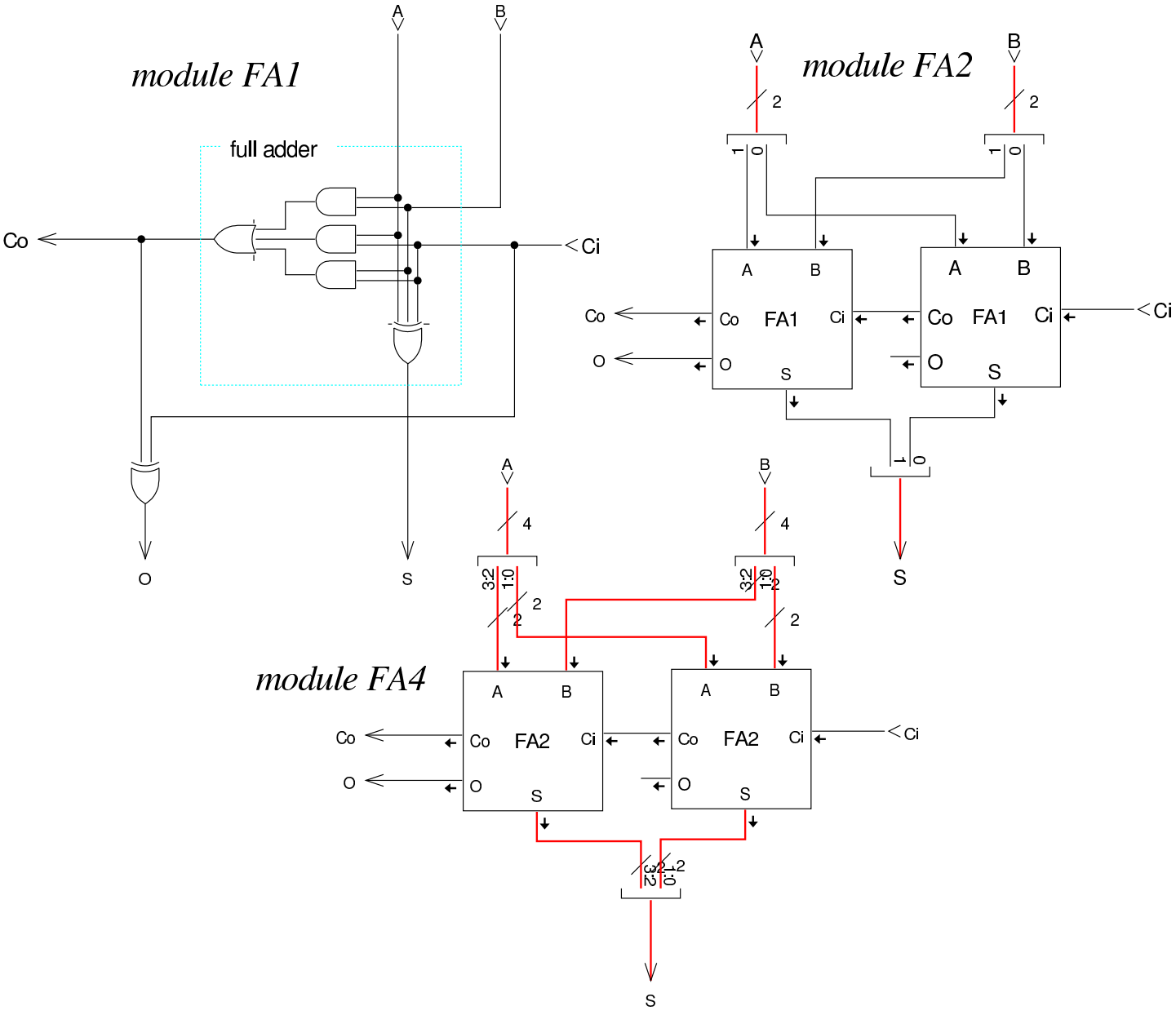


Figure 12.20. Module **SH**: bit shift.

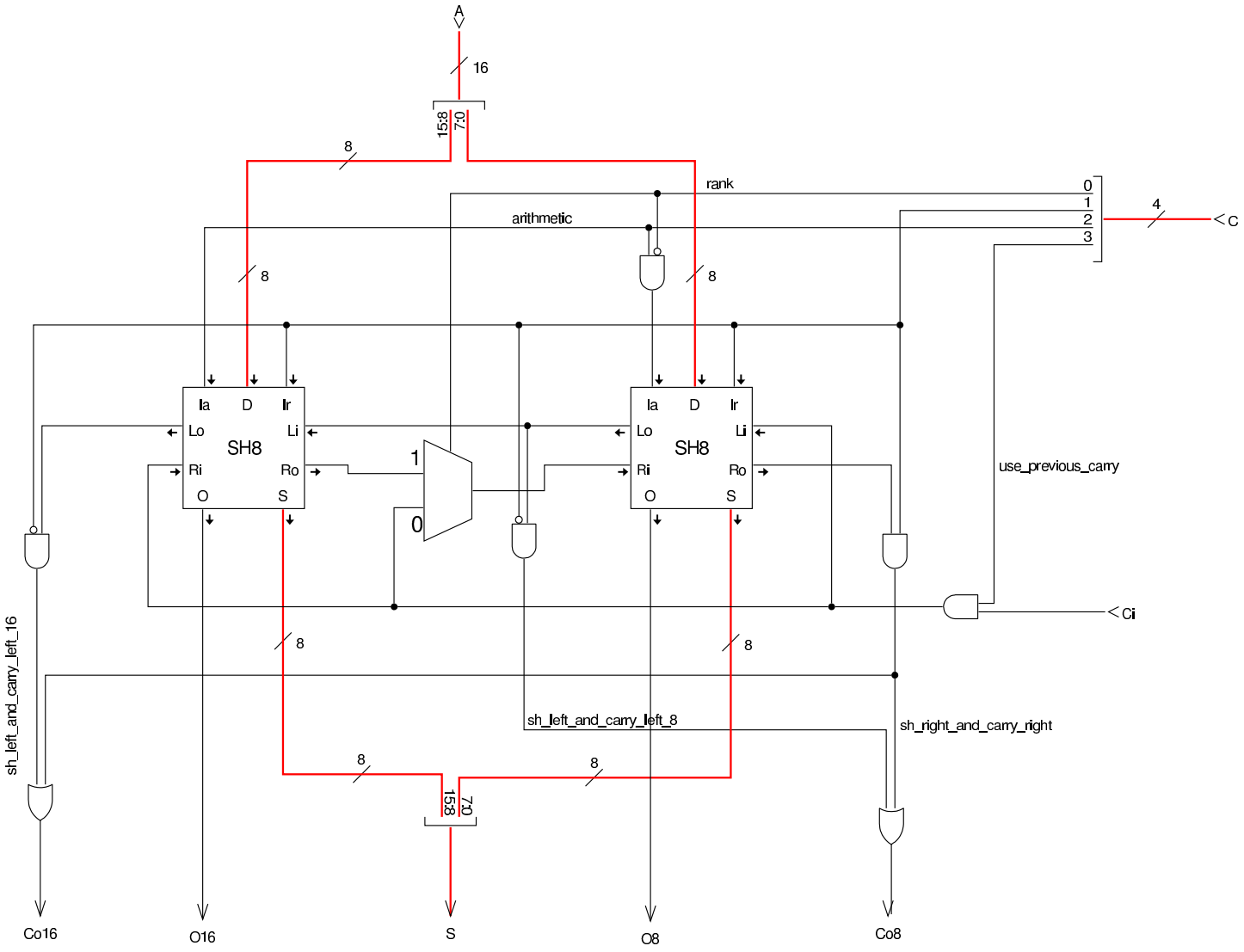
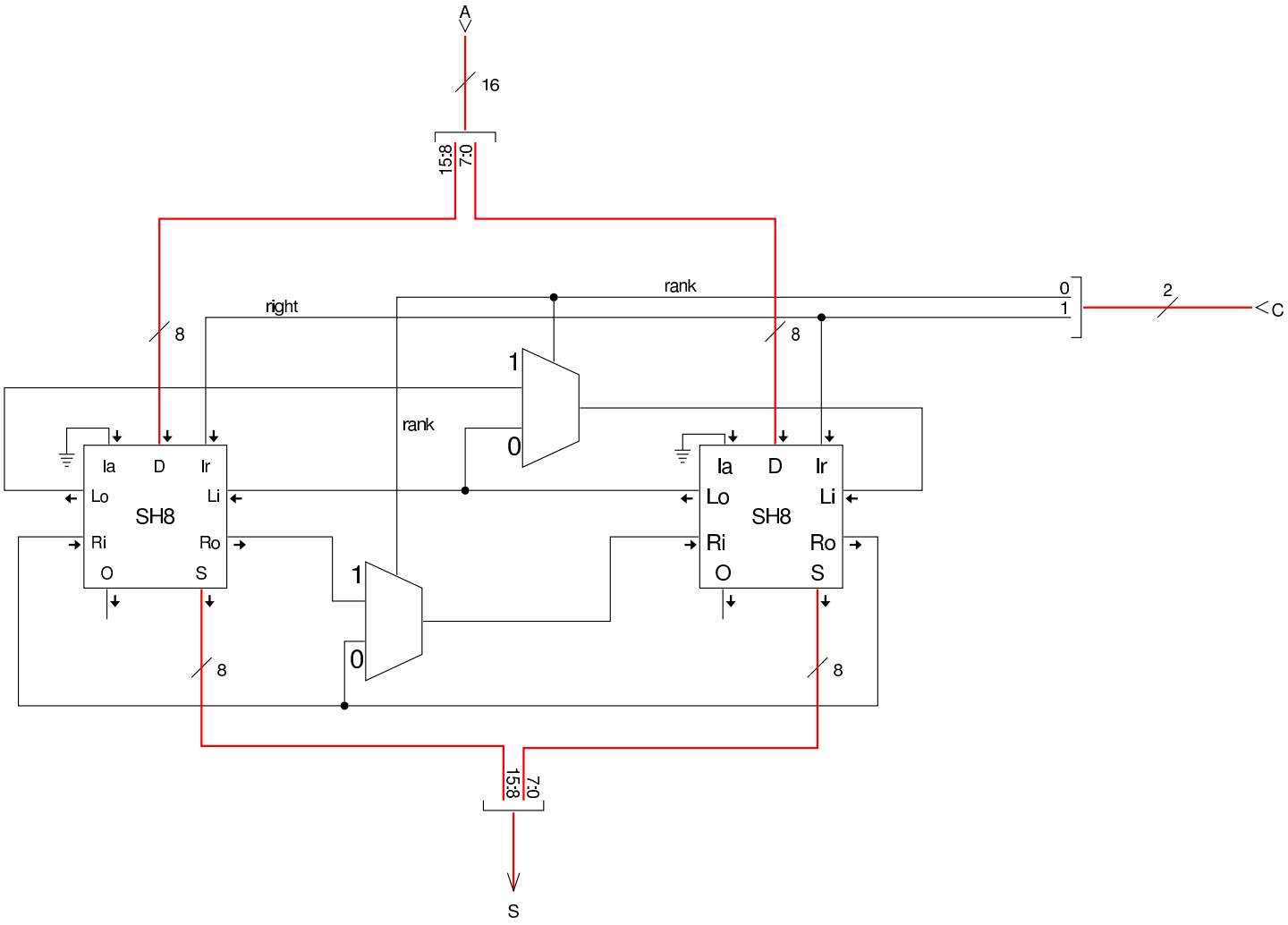
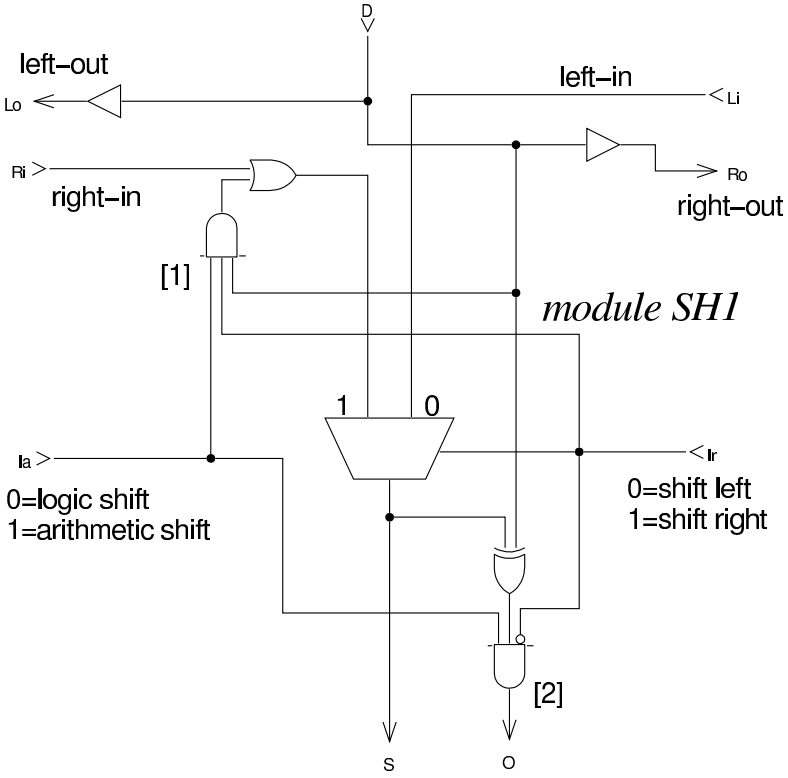


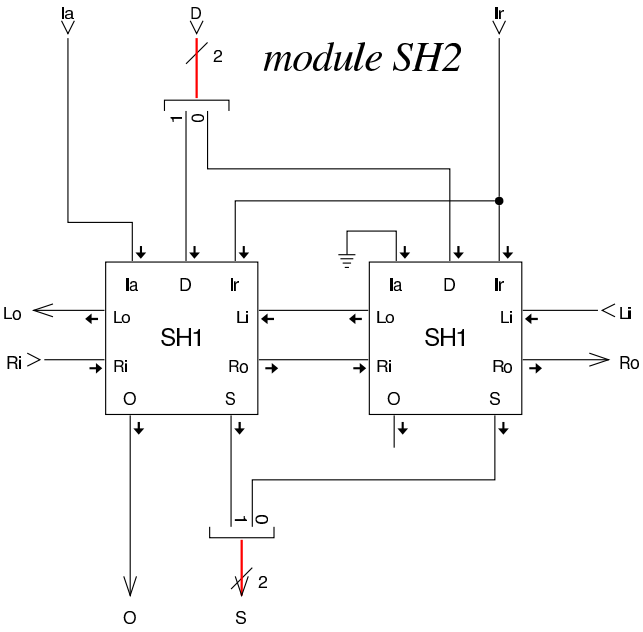
Figure 12.21. Module **ROT**: bit rotation.

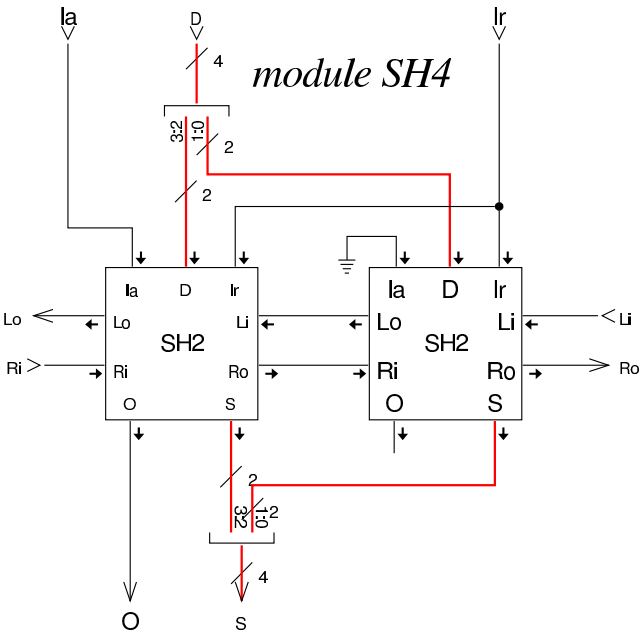


The modules **SH*n*** are one-bit logic or arithmetic shifter. It all starts from module **SH1**, as the following figure shows.

Figure 12.22. Module **SH*n***: one-bit logic or arithmetic shift.



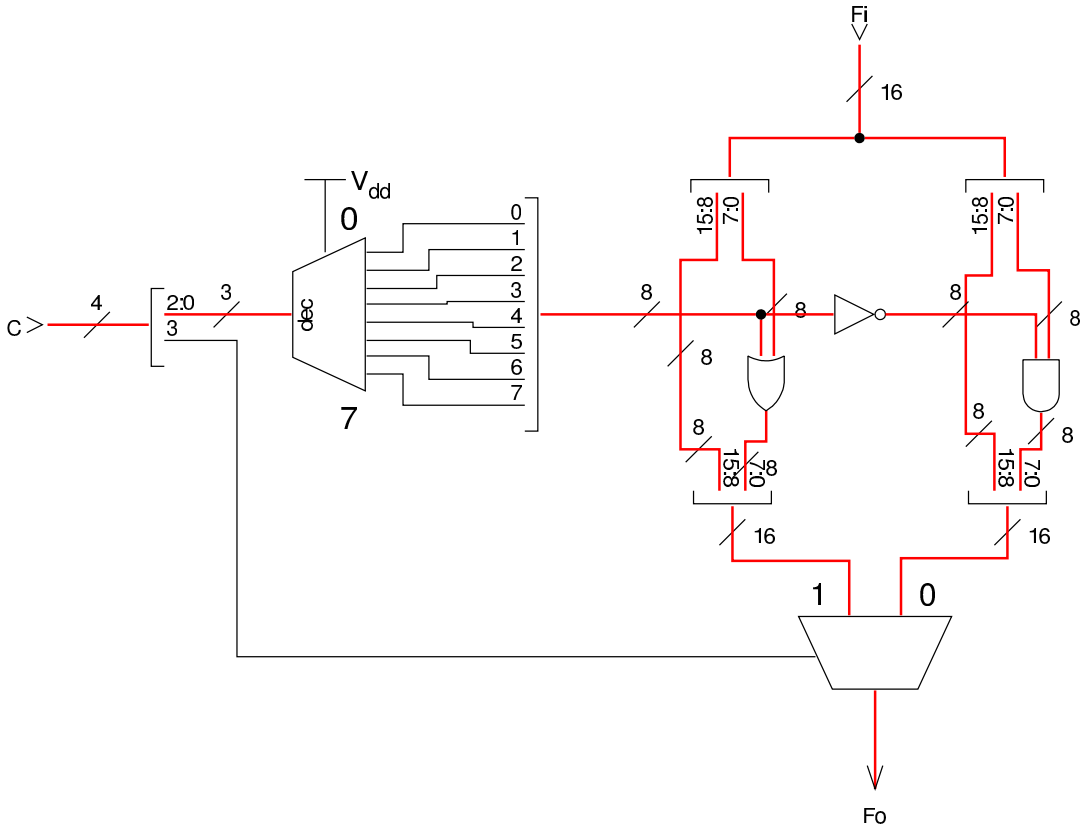




[1] Arithmetic shift conditions:
- the requested shift is arithmetic;
- the requested shift is to the right;
- the most significant bit value is one.

[2] Overflow conditions:
- the requested shift is arithmetic;
- the requested shift is to the left;
- the most significant bit value is changed.

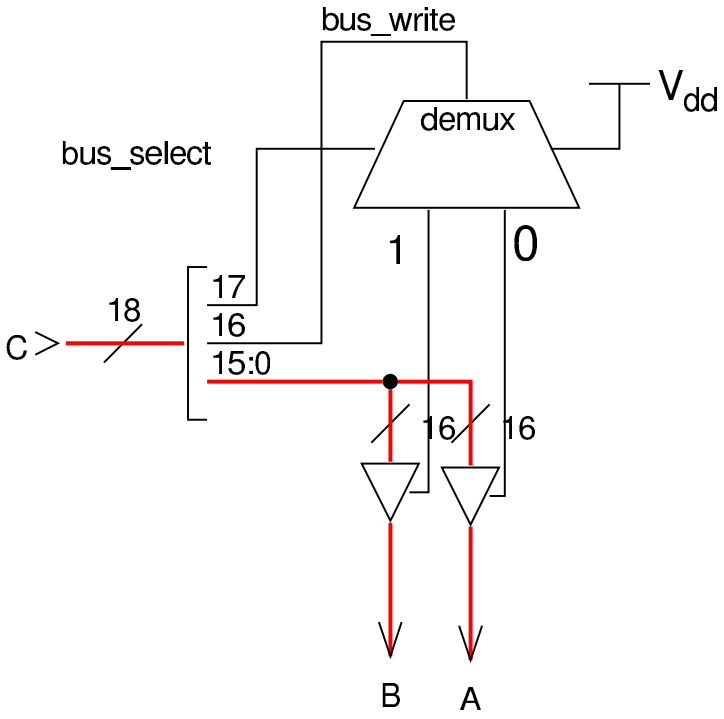
Figure 12.23. Module **FSR**: flags set-reset.



12.5 Module "BUS"

The module **BUS** is used to let the control unit to put a value inside the data-bus. This module **BUS** is the same as the previous version.

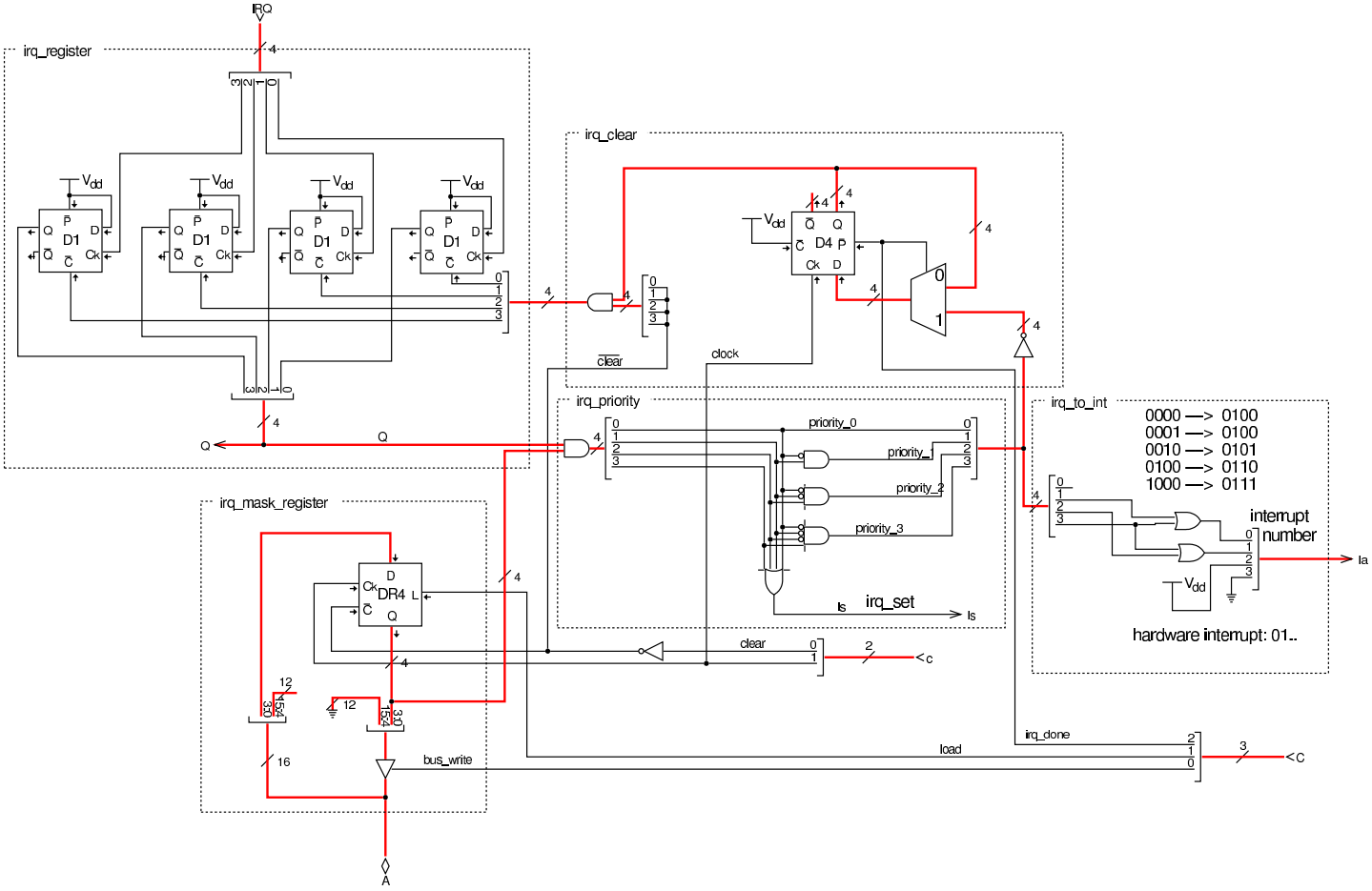
Figure 12.24. Module **BUS**.



12.6 Module "IRQ"

The **IRQ** module is the same as the previous version. All the details about the module functionality should be found at the previous version description.

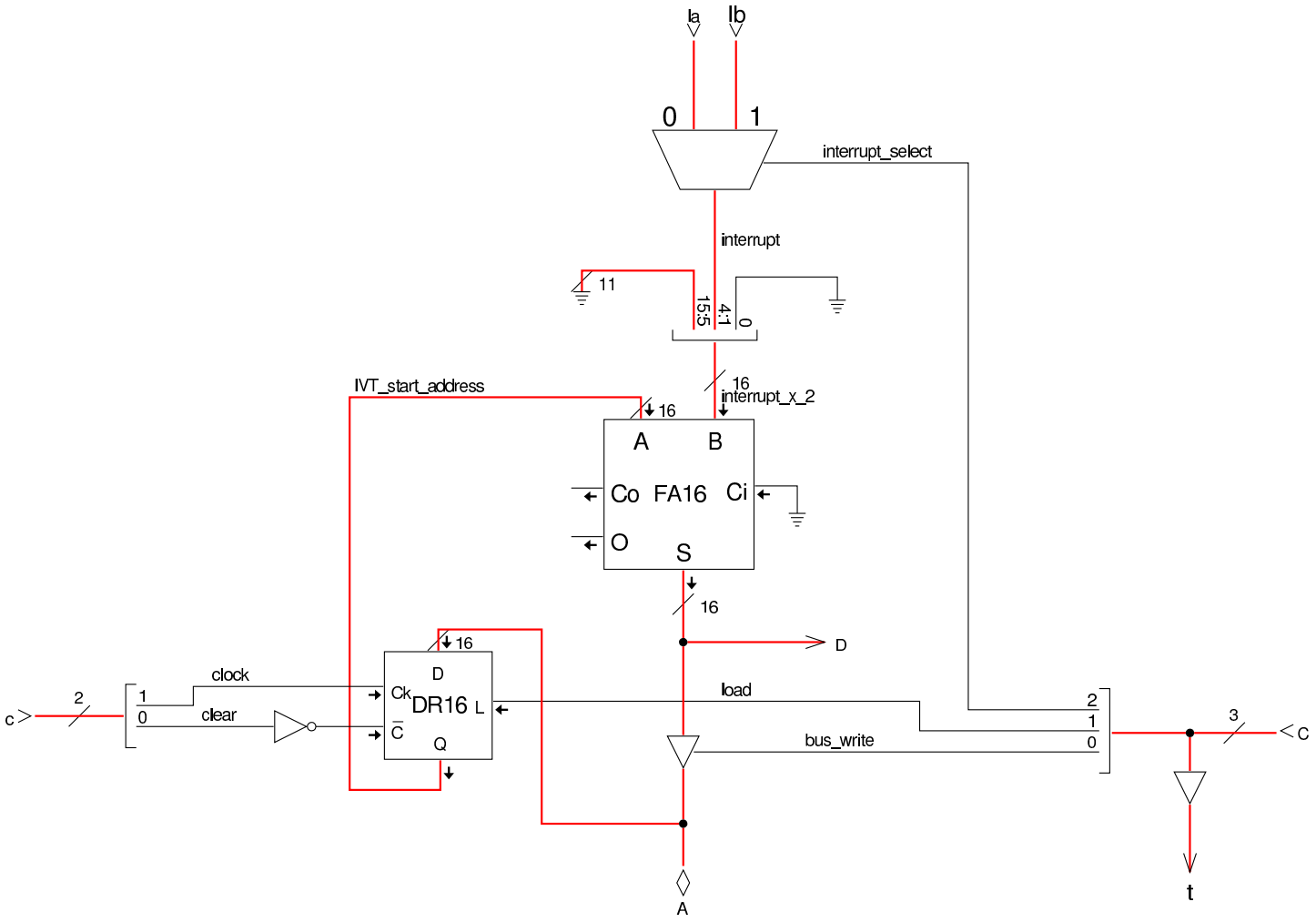
Figure 12.25. Module **IRQ**.



12.7 Module "IVT"

The **IVT** module is the same as the previous version. All the details about the module functionality should be found at the previous version description.

Figure 12.26. Module **IVT**.



12.8 Module "RTC": real time clock

«

The **RTC** module (*real time clock*) is just the same as the previous version: it generates a 1 Hz impulse, producing the hardware interrupt IRQ0. All the details about the module functionality should be found at the previous version description.

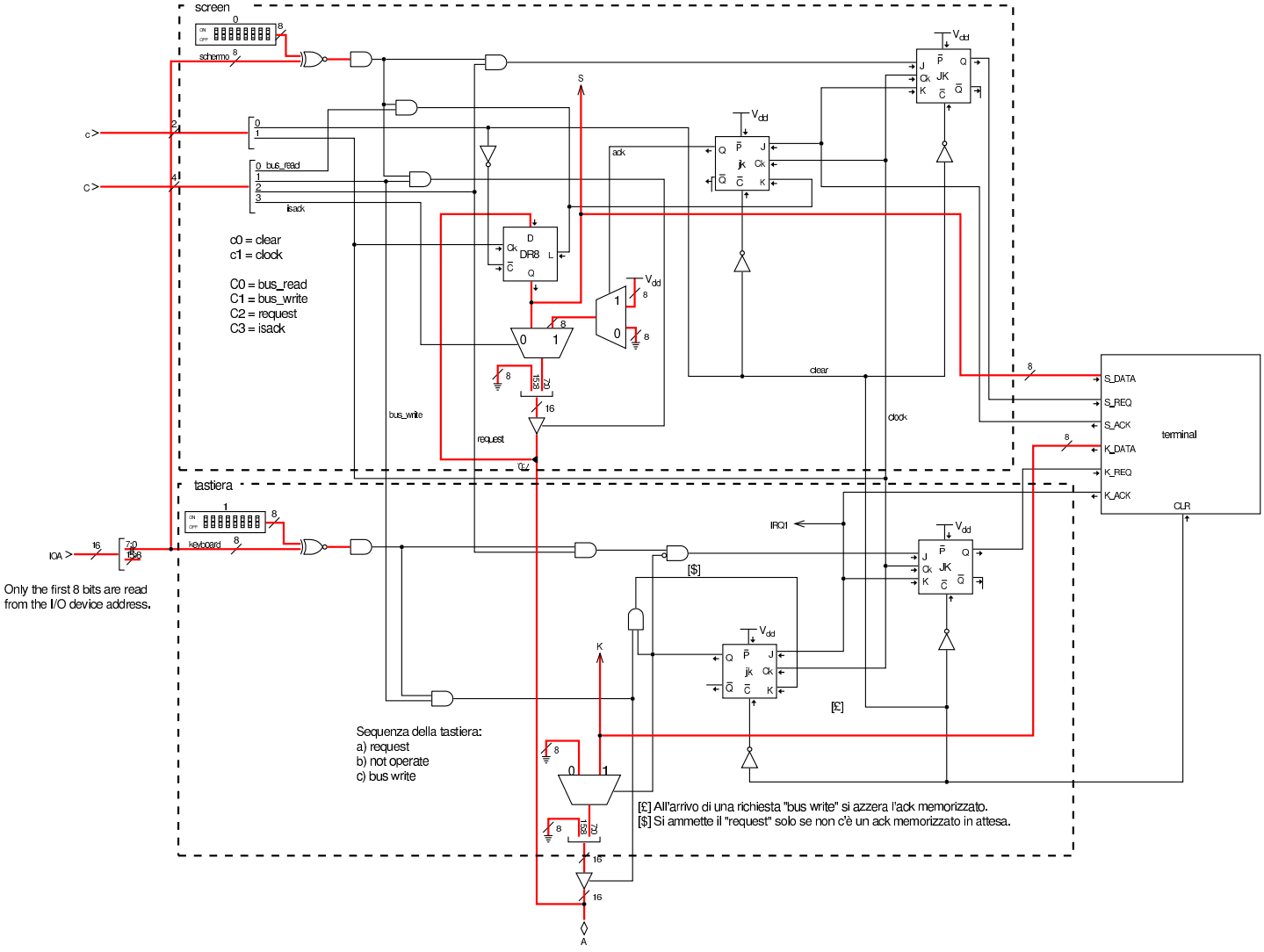
Listing 12.27. Module **RTC**: Verilog declaration.

```
module RTC (T);
  output T;
  reg p;
  always
    begin
      p = 0;
      $tkg$wait (500);
      p = 1;
      $tkg$wait (500);
    end
  assign T = p;
endmodule
```

12.9 Module "TTY"

The **TTY** module is just the same as the previous version: it is a textual screen-keyboard terminal interface. All the details about the module functionality should be found at the previous version description.

Figure 12.28. Module **TTY**.



Listing 12.29. Module **terminal**: Verilog code.

```
module terminal(K_DATA, K_REQ, K_ACK, S_DATA, S_REQ, S_ACK, CLR);

output K_ACK;
output S_ACK;
output [7:0] K_DATA;
input [7:0] S_DATA;
input K_REQ;
```

```
input S_REQ;
input CLR;
reg k_ready;
reg [7:0] key;
reg s_ready;

  initial
    begin
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  always
    begin
      @(posedge CLR)
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  initial $tkg$post("TERMINAL","%m");

  always
    begin
      @ (posedge K_REQ);
      # 5;
      key = $tkg$recv("%m.KD");
      # 5;
      k_ready = 1'b1;
      # 5;
      @ (negedge K_REQ);
      # 5;
      k_ready = 1'b0;
    end

  always
```

```
    begin
      @(posedge S_REQ);
      # 5;
      $tkg$send("%m.SD",S_DATA);
      # 5;
      s_ready = 1'b1;
      # 5;
      @(negedge S_REQ);
      # 5;
      s_ready = 1'b0;
    end

  assign S_ACK = s_ready;
  assign K_DATA = key;
  assign K_ACK = k_ready;

endmodule
```

Listing 12.30. File ‘share/tkgate/vpd/terminal.tcl’ for TCL interface.

```
image create bitmap txtcurs -file "$bd/txtcurs.b"

VPD::register TERMINAL
VPD::allow TERMINAL::post
VPD::allow TERMINAL::data

namespace eval TERMINAL {
    # Public variables declarations: the variables $terminal_...
    # are arrays of which only the element $n is used;
    # that element identifies uniquely the working interface instance.
    variable terminal_w
    variable terminal_pos
    #
    variable KD
    # Function requested by TKGate to create the interface.
    proc post {n} {
        variable terminal_w
        variable terminal_pos
        # Create the window and save the object element in a $terminal_w array element.
        set terminal_w($n) [VPD::createWindow "TERMINAL $n" -shutdowncommand "TERMINAL::unpost $n"]
        # For convenience, copy the object reference inside the local
        # variable $w; then, the variable $w will be used as a reference to the object.
        set w $terminal_w($n)
        text $w.txt -state disabled
        pack $w.txt
```

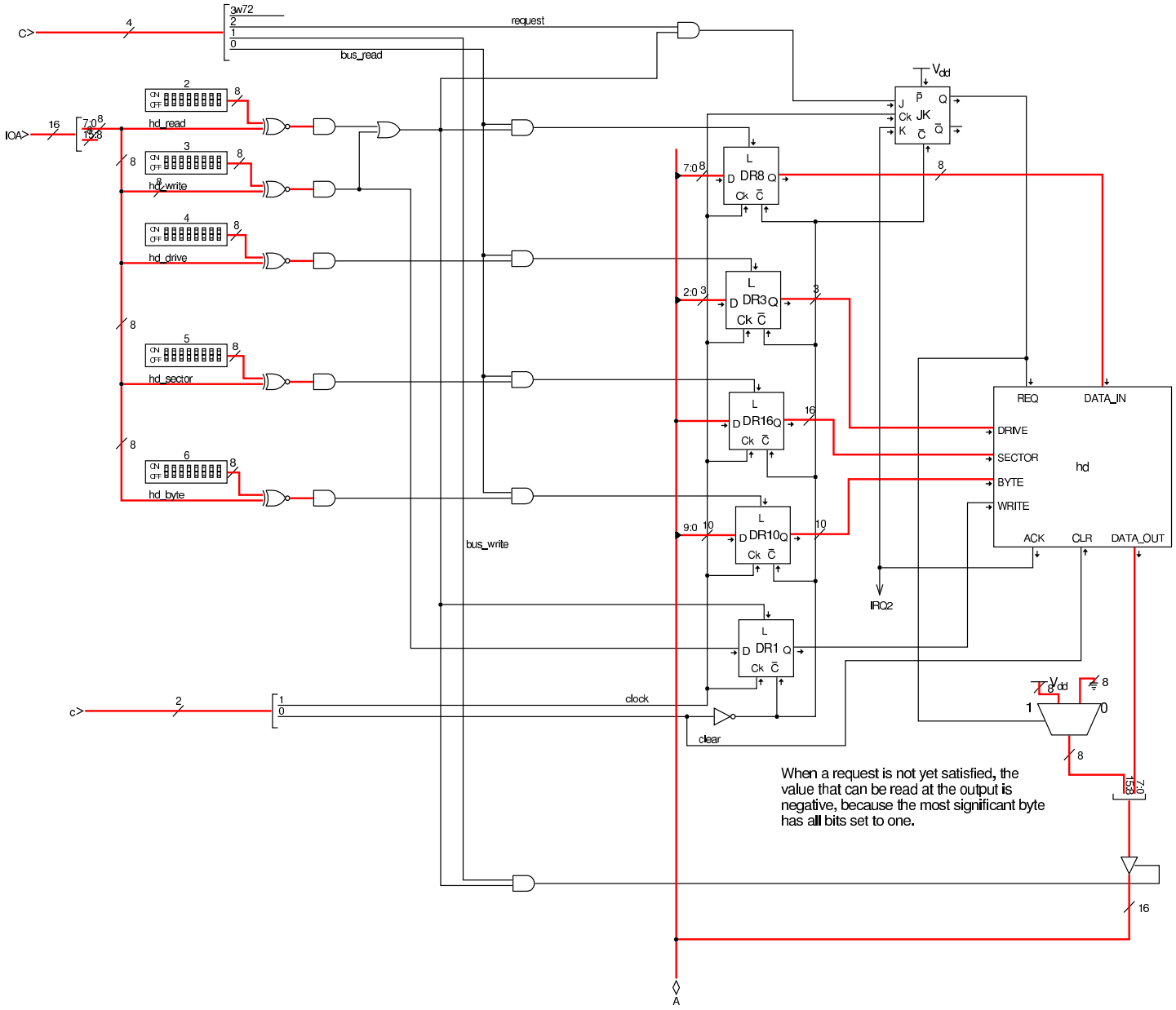
```
        # Put the cursor at the end of the displayed text.
        $w.txt image create end -image txtcurs
        # Bind the keyboard input, related to the object represented by
        # $terminal_w($n), to the function sendChar.
        bind $w <KeyPress> "TERMINAL::sendChar $n \"%A\""
        # Open a reading channel, named «SD» (screen data),
        # and associate it to the function «data»; moreover, open a
        # writing channel, named «KD» (keyboard data).
        if {[info exists ::tkgate_isInitialized]} {
            VPD::outsignal $n.KD TERMINAL::KD($n)
            VPD::insignal $n.SD -command "TERMINAL::data $n" -format %d
        }
        # Reset the character count, used to count the characters displayed
        # on screen.
        set terminal_pos($n) 0
    }
    # Function that receives the typing and put it into the
    # channel «KD», related to the current interface instance.
    proc sendChar {n key} {
        variable KD
        if { [string length $key ] == 1 } {
            binary scan $key c c
            set TERMINAL::KD($n) $c
        }
    }
    # Function that TKGate requires to destroy the interface.
    proc unpost {n} {
        variable terminal_w
        variable terminal_pos
        destroy $terminal_w($n)
        destroy $terminal_pos($n)
        unset terminal_w($n)
        unset terminal_pos($n)
    }
    # Function used to get the data to display on screen.
    proc data {n c} {
        variable terminal_w
        variable terminal_pos
        # For convenience, copy the object reference representing the
        # interface, inside the variable $w.
        set w $terminal_w($n)
        catch {
            # The variable $c contains the character to display.
            if { $c == 7 } {
                # BEL
                bell
                return
            } elseif { $c == 127 || $c == 8 } {
                # DEL | BS
                if { $terminal_pos($n) > 0 } {
                    # Delete the last displayed character, but only if the
                    # characters counter is greater than zero, otherwise
```

```
                    # the cursor would disappear and the next characters
                    # would be located in an unvisible screen area.
                    $w.txt configure -state normal
                    $w.txt delete "end - 3 chars"
                    $w.txt see end
                    $w.txt configure -state disabled
                    set terminal_pos($n) [expr {$terminal_pos($n) - 1}]
                }
                return
            } elseif { $c == 13 } {
                # Convert CR to LF.
                set c 10
            }
            # Convert the character number into a visible symbol.
            set x [format %c $c]
            # Display the symbol.
            $w.txt configure -state normal
            $w.txt insert "end - 2 chars" $x
            $w.txt see end
            $w.txt configure -state disabled
            # Update the displayed characters counter.
            set terminal_pos($n) [expr {$terminal_pos($n) + 1}]
        }
    }
}
```

12.10 Module "HDD"

The **HDD** module is just the same as the previous version: it is a simulated mass-memory drive whith the ability to handle eight units. All the details about the module functionality should be found at the previous version description.

Figure 12.31. Module **HDD**.



Listing 12.32. Verilog code describing the module **hd**.

```
module hd(DRIVE, SECTOR, BYTE, WRITE, DATA_IN, DATA_OUT, REQ, ACK, CLR);
input [2:0] DRIVE;
input WRITE, REQ, CLR;
input [15:0] SECTOR;
input [9:0] BYTE;
input [7:0] DATA_IN;
output [7:0] DATA_OUT;
output ACK;
//
```

```
integer _data_out;
integer _ack;
//
reg [7:0] buffer[0:1023];
reg [8*24-1:0] filename = "hd0_sector_000000000.mem";
//
integer i;
integer sector_8;
integer sector_7;
integer sector_6;
integer sector_5;
integer sector_4;
integer sector_3;
integer sector_2;
integer sector_1;
integer sector_0;
integer x;
//
  initial
    begin
      for (i=0; i<1024; i=i+1)
        begin
          //
          // Initial buffer reset with 00.
          //
          buffer[i] = 8'h00;
        end
      _ack = 0;
      _data_out = 0;
      x = 0;
    end
  //
  always
    begin
      @(posedge CLR)
      _ack = 0;
      _data_out = 0;
      x = 0;
    end
  //
```

```
  //
  //
  always
    begin
      //
      // Start after a positive edge from REQ!.
      //
      @(posedge REQ);
      # 10;
      //
      // Define the sector file name.
      //
      x = SECTOR;
      sector_0 = x%10;
      x = x/10;
      sector_1 = x%10;
      x = x/10;
      sector_2 = x%10;
      x = x/10;
      sector_3 = x%10;
      x = x/10;
      sector_4 = x%10;
      x = x/10;
      sector_5 = x%10;
      x = x/10;
      sector_6 = x%10;
      x = x/10;
      sector_7 = x%10;
      x = x/10;
      sector_8 = x%10;
      //
      // The string starts from the right side and continues
      // to the left!
      //
      filename[12*8+7:12*8] = sector_8 + 8'd48;
      filename[11*8+7:11*8] = sector_7 + 8'd48;
      filename[10*8+7:10*8] = sector_6 + 8'd48;
      filename[9*8+7:9*8] = sector_5 + 8'd48;
      filename[8*8+7:8*8] = sector_4 + 8'd48;
      filename[7*8+7:7*8] = sector_3 + 8'd48;
```

```
        filename[6*8+7:6*8] = sector_2 + 8'd48;
        filename[5*8+7:5*8] = sector_1 + 8'd48;
        filename[4*8+7:4*8] = sector_0 + 8'd48;
        //
        filename[21*8+7:21*8] = DRIVE + 8'd48;
        //
        if (WRITE)
          begin
            //
            // Put data inside the buffer.
            //
            buffer[BYTE] = DATA_IN;
            //
            // Save the buffer to disk.
            // Please remember that $writememh() must be enabled inside
            // TKGate configuration!
            //
            $writememh(filename, buffer);
            //
            // Return the same data read.
            //
            _data_out = buffer[BYTE];
          end
        else
          begin
            //
            // Get data from disk to the buffer.
            //
            $readmemh(filename, buffer);
            //
            // Return the data required.
            //
            _data_out = buffer[BYTE];
          end
        //
        // Acknowledge.
        //
        _ack = 1;
        //
        // Wait the end of request (the negative edge)
```

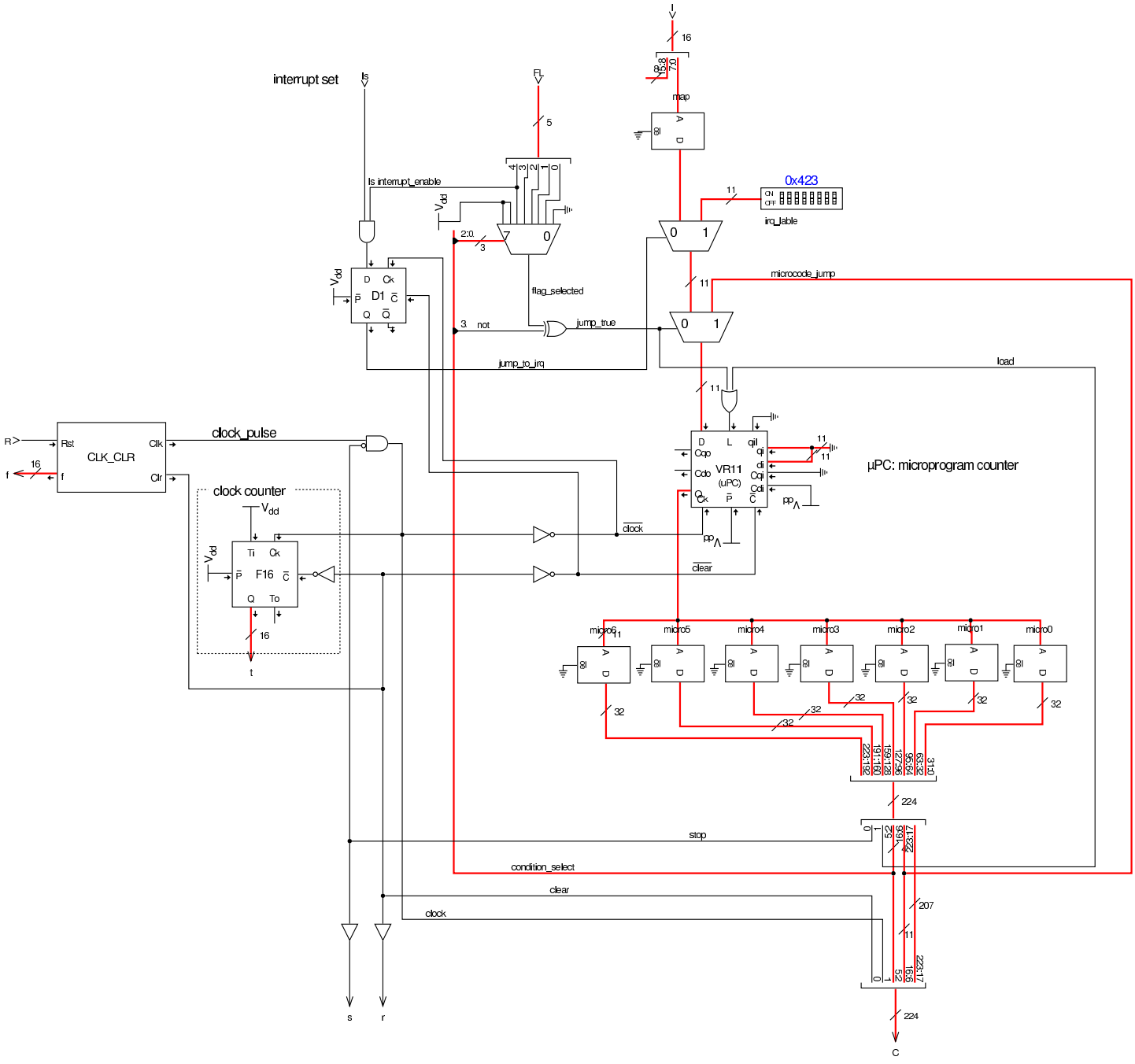
```
      // before restarting the loop.
      //
      @(negedge REQ);
      # 10;
      //
      // Now become ready again.
      //
      _ack = 0;
    end
    //
  assign DATA_OUT = _data_out;
  assign ACK = _ack;
  //
endmodule
```

12.11 Module "CTRL"

«

The module **CTRL** is the same as almost the previous version, except for the name of the microprogram counter (now μ**PC**) and its size (11 bit).

Figure 12.33. Module **CTRL**.



12.11.1 Modules "F*n*"

The modules **F*n*** frequency divisors, made the same way as for the previous version.

Figure 12.34. Modules **D1** and **T1**: building a T flip-flop from a D flip-flop.

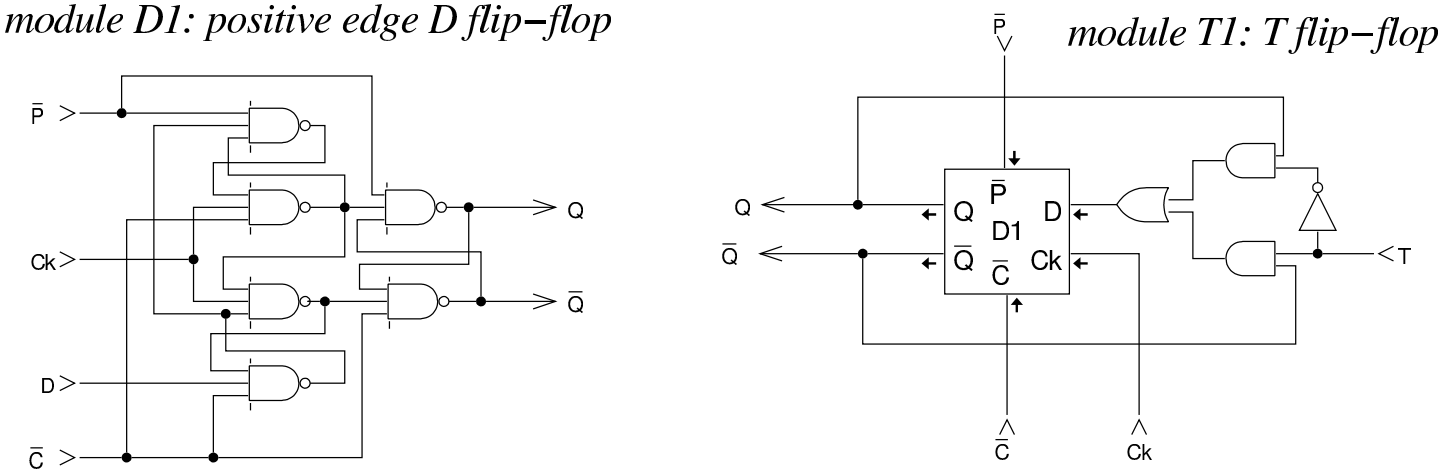
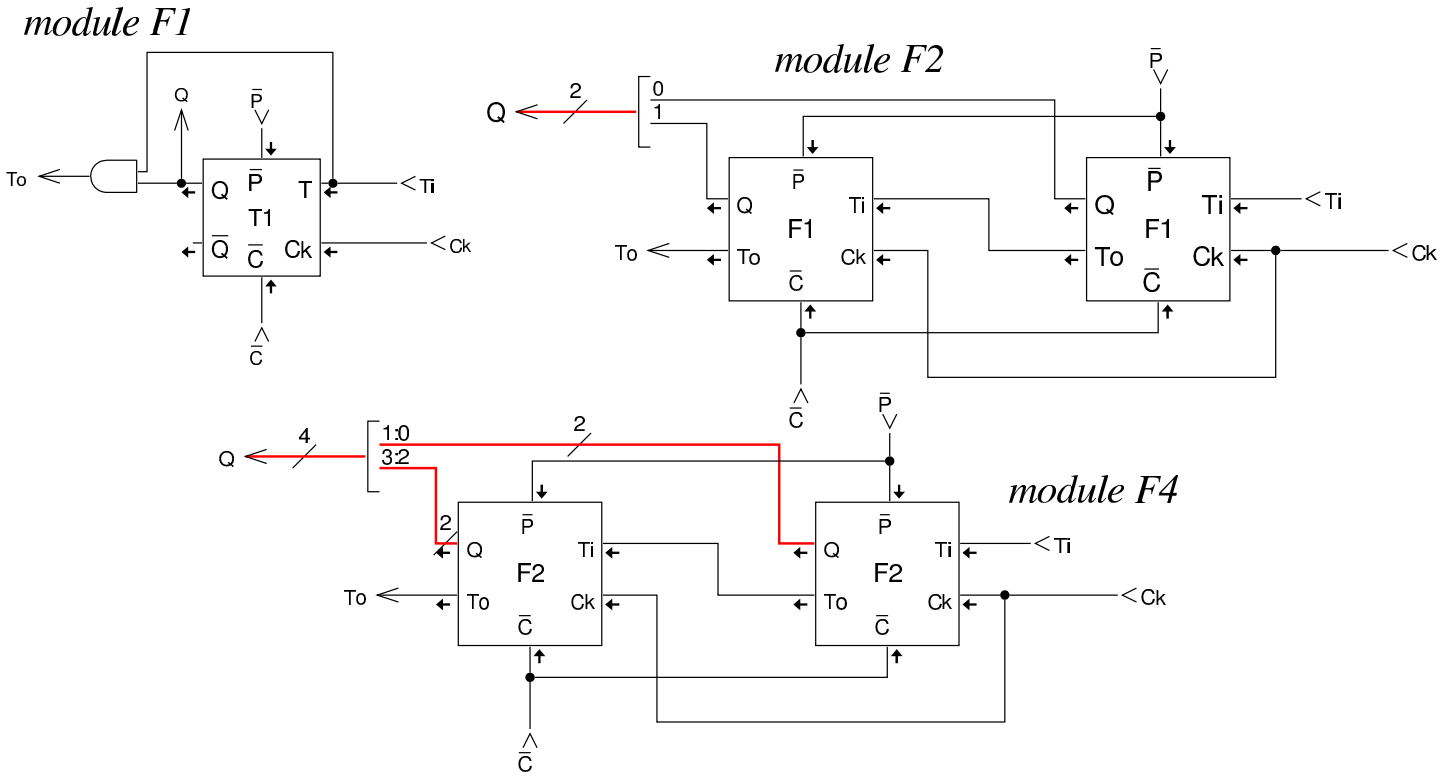


Figure 12.35. Building the **F*n*** modules.



12.11.2 Module "CLK_CLR"

The module **CLK_CLR** is responsible for the clock pulse generation and for the reset line, which is to be synchronized with the clock. The clock frequency may be controlled by the dip-switch at the top. It is important to remind that the control unit receives an inverted clock pulse, because the microcode data should be ready before the clock positive edge. This one is just the same as the previous version.

Figure 12.36. Module **CLK_CLR**.

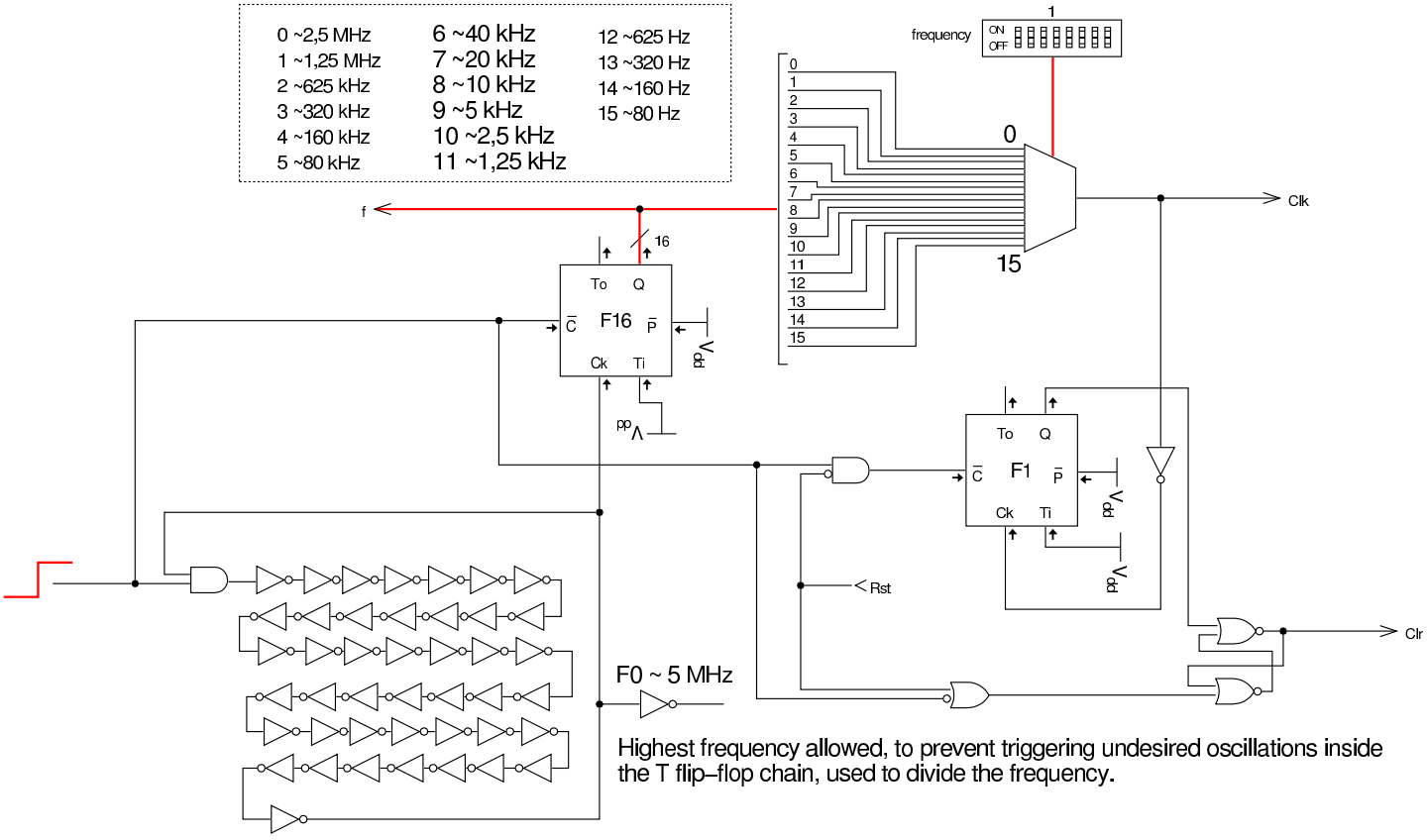


Figure 12.37. Module **one_up**, used to start the oscillation inside the module **CLK_CLR**.

```
module one_up #(.W(1000)) (Z);
  output Z;
  reg Z;

  initial
    begin
      Z = 1'b0;
      $tkg$wait(W);
      Z = 1'b1;
    end

endmodule
```

12.12 Memory and microcode fields

«

The memory for this version of the project are described by the following lines of TKGate code. There are many blocks of memory for the microcode word, because every component connected to the data-bus can be controlled independently, as it is for the previous version of the project.

Listing 12.38. Memory banks description for TKGate.

```
map       bank[9:0]     ctrl.map;
microcode bank[31:0]    ctrl.micro0;
microcode bank[63:32]   ctrl.micro1;
microcode bank[95:64]   ctrl.micro2;
microcode bank[127:96]  ctrl.micro3;
microcode bank[159:128] ctrl.micro4;
microcode bank[191:160] ctrl.micro5;
microcode bank[223:192] ctrl.micro6;
macrocode bank[15:0]    ram;
```

Listing 12.39. Fields of the microcode word, for TKGate.

```
field ctrl[1:0]    = {nop=0, stop=1, load=2};
field jump[5:2]    = {false=0, carry_t=1, zero_t=2, negative_t=3,
                      overflow_t=4, irq_enabled_t=5, true=6,
                      carry_f=9, zero_f=10, negative_f=11,
                      overflow_f=12, irq_enabled_f=13};
field addr[16:6]   = {ciao=0};
//
field mar[22:17]   = {load=0x0001, ram_write=0x0002,
                      rank8=0, rank16=0x0004,
                      p=0, i=8, j=16, s=24, t=32};
//
field mdr[28:23]   = {a=0x0000, b=0x0001, bw=0x0002,
                      rank8u=0, rank16=0x0004, rank8s=0x0008,
                      load=0x0010, ram_read=0x0020};
//
field mir[29:29]   = {ram_read=0x0001};
//
field pc[41:30]    = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                      qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                      qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
```

```
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
// Macro for instruction fetch.
// MAR <= PC++, MIR <= RAM
// f1: mar=p mar=rank8 mar=load pc=dp1 mir=ram_read;
//
field fetch[41:17] = {f1=0x201001};
//
field i[53:42]     = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field j[65:54]     = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field sp[77:66]    = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
```

```
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field tmp[89:78]    = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field fl[101:90]    = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field c[113:102]    = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field alu[122:114] = {rank8u=0, rank8s=1, rank16=2,
                        a=0, and=4, or=8, xor=12,
                        nxor=16, nor=20, nand=24, not=28,
                        add=64, sub=68, addc=72, subb=76,
                        lshl=128, lshr=132, ashl=136, ashr=140,
                        rotcl=144, rotcr=148,
                        rotl=192, rotr=196,
```

```
                        clearc=256, clearz=260, clearn=264,
                        clearo=268, cleari=272,
                        setc=288, setz=292, setn=296,
                        seto=300, seti=304};
//
field a[134:123]    = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field b[146:135]    = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field bp[158:147]   = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field bus[176:159] = {bw=0x10000, a=0, b=0x20000};
//
field ioa[188:177] = {a=0x0000, b=0x0001, bw=0x0002, load=0x0004,
                        qp1=0x0008, qp2=0x0010, qp3=0x0018, qp4=0x0020,
                        qp5=0x0028, qp6=0x0030, qp7=0x0038, qm8=0x0040,
```

```
                        qm7=0x0048, qm6=0x0050, qm5=0x0058, qm4=0x0060,
                        qm3=0x0068, qm2=0x0070, qm1=0x0078, qup=0x0080,
                        dp1=0x0100, dp2=0x0200, dp3=0x0300, dp4=0x0400,
                        dp5=0x0500, dp6=0x0600, dp7=0x0700, dm8=0x0800,
                        dm7=0x0900, dm6=0x0A00, dm5=0x0B00, dm4=0x0C00,
                        dm3=0x0D00, dm2=0x0E00, dm1=0x0F00 };
//
field ivt[191:189] = {bw=1, load=2, int_a=0, int_b=4};
//
field irq[194:192] = {bw=1, mask_load=2, irq_done=4};
//
field ioc[198:195] = {load=1, bw=2, req=4, isack=8};
```

12.13 Opcodes

The opcodes are exactly the same as the previous version. For convenience, the following table shows the opcode organization.

Table 12.40. Opcodes.

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0x00	00......	cp % %		%			%					
0x40	010000..	cp n %						A, B, I, J		const16		
0x44	010001..	cp (%) %						(I, J)	A, B			
0x48	010010..	cp % (%)						A, B	(I, J)			
0x4C	0100110.	cp n (%)							(I, J)	const16		
0x4E	0100111.	cp (%) (%)							(I, J)			
0x50	010100..	cp8 n %						A, B, I, J		const8		
0x54	010101..	cp8 (%) %						(I, J)	A, B			
0x58	010110..	cp8 % (%)						A, B	(I, J)			
0x5C	0101110.	cp8 n (%)							(I, J)	const8		
0x5E	0101111.	cp8 (%) (%)							(I, J)			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0x60	01100...	push %					%					
0x68	01101...	push8 %					%					
0x70	01110...	pop %					%					
0x78	01111...	pop8 %					%					
0x80	10000000	push n								const16		
0x81	10000001	push8 n								const8		
0x82	1000001.	push (%)							(I, J)			
0x84	1000010.	push8 (%)							(I, J)			
0x86	1000011.	pop (%)							(I, J)			
0x88	1000100.	pop8 (%)							(I, J)			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0x90	10010...	jump %flag 0|1 #ref					%flag		TRUE	#ref		
0x98	1001100.	equal, not							NOT			
0x9A	1001101.	and							NOT			
0x9C	1001110.	or							NOT			
0x9E	1001111.	xor							NOT			
0xA0	1010000.	add, sub							sub			
0xA2	1010001.	addc, subb							sub			
0xA4	1010010.	lsh							right			
0xA6	1010011.	ash							right			
0xA8	1010100.	rotc							right			
0xAA	1010101.	rot							right			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0xAC	1010110.	add8, sub8							sub			
0xAE	1010111.	add8c, sub8b							sub			
0xB0	1011000.	lsh8							right			
0xB2	1011001.	ash8							right			
0xB4	1011010.	rot8c							right			
0xB6	1011011.	rot8							right			
0xB8	101110..	cast						sign	A, B			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0xBC	1011110.	equals, nots							NOT			
0xBE	1011111.	ands							NOT			
0xC0	1100000.	ors							NOT			
0xC2	1100001.	xors							NOT			
0xC4	1100010.	adds, subs							sub			
0xC6	1100011.	addcs, subbs							sub			
0xC8	1100100.	lshls, lshrs							right			
0xCA	1100101.	ashls, ashrs							right			
0xCC	1100110.	rotcls, rotcrs							right			
0xCE	1100111.	rotls, rotrs							right			

		byte 0								byte 1	byte 2	byte 3
		b7	b6	b5	b4	b3	b2	b1	b0			
0xD0	1101000.	add8s, sub8s							sub			
0xD2	1101001.	add8cs, sub8bs							sub			
0xD4	1101010.	lsh8ls, lsh8rs							right			
0xD6	1101011.	ash8ls, ash8rs							right			
0xD8	1101100.	rot8cls, rot8crs							right			
0xDA	1101101.	rot8ls, rot8rs							right			
0xDC	1101110.	casts							sign			

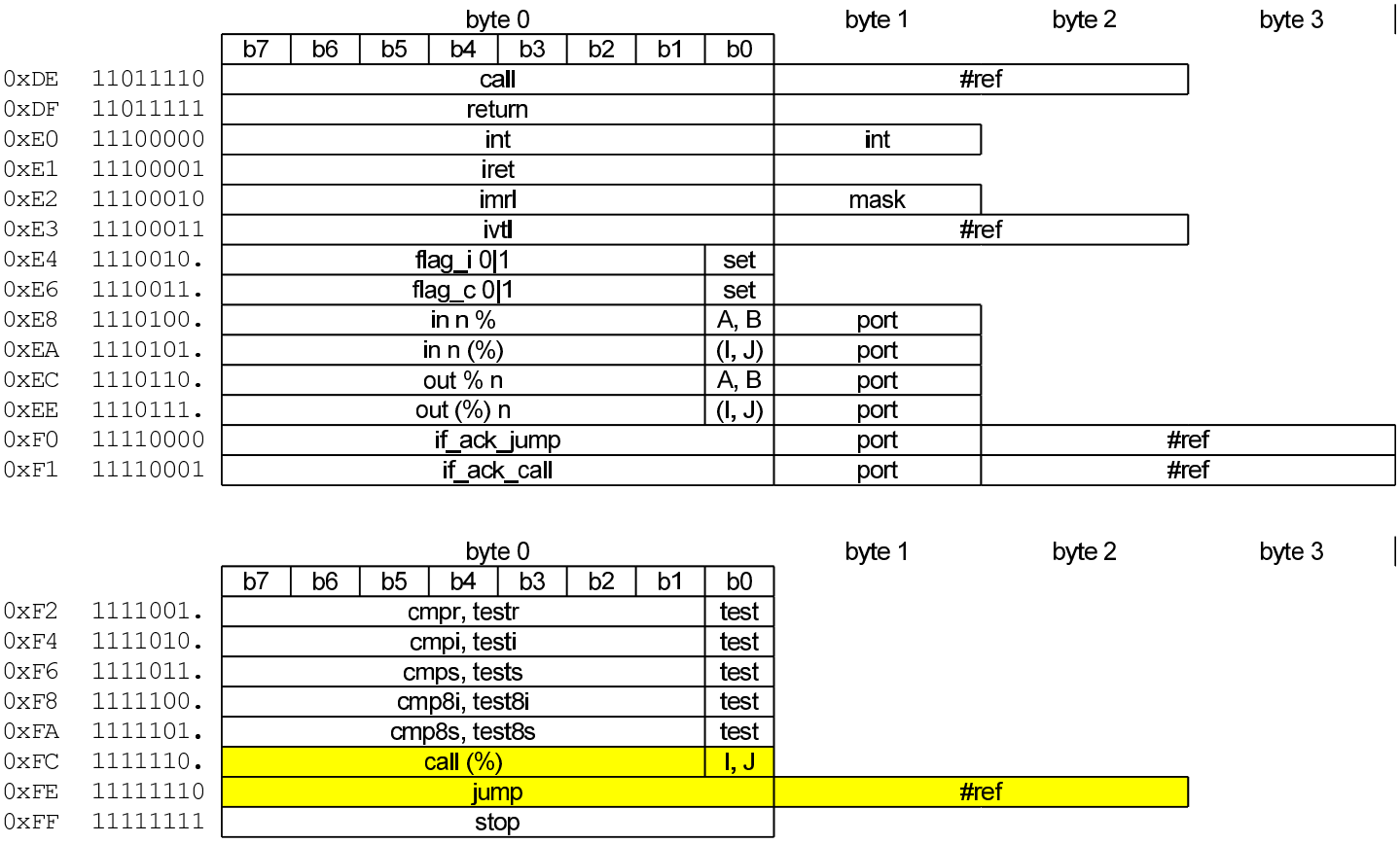


Table 12.41. Macrocode syntax: it is the same as the previous version.

Macrocode syntax	Description
`nop`	Not operate.
`cp %`***src***`, %`***dst***	Copy the ***src*** register content inside the ***dst*** register. Allowed registers are: ***I***, ***J***, ***A***, ***B***, ***BP***, ***SP***, ***C***, ***FL***.
`cp` ***n16***`, %`***dst***	Assign a 16-bit constant to a register. Allowed destination registers are: ***I***, ***J***, ***A***, ***B***.
`cp (%I|%J), %A|%B`	Assign to register ***A*** or ***B*** the 16-bit value contained in memory at the address already available from register ***I*** or ***J***.
`cp %A|%B, (%I|%J)`	Copy in memory the 16-bit value of register ***A*** or ***B***, to the location represented by the register ***I*** or ***J***.
`cp` ***n16***`, (%I|%J)`	Assign in memory the 16-bit constant, to the location represented by the register ***I*** or ***J***.

Macrocode syntax	Description
`cp (%I)`	Copy the 16-bit memory content at the address represented by register ***I***, to the memory location represented by the register ***J***.
`cp (%J)`	Copy the 16-bit memory content at the address represented by register ***J***, to the memory location represented by the register ***I***.
`cp8` ***n8***`, %`***dst***	Assign a 8-bit constant to a register. Allowed destination registers are: ***I***, ***J***, ***A***, ***B***.
`cp8 (%I|%J), %A|%B`	Assign to register ***A*** or ***B*** the 8-bit value contained in memory at the address already available from register ***I*** or ***J***.
`cp8 %A|%B, (%I|%J)`	Copy in memory the 8-bit value of register ***A*** or ***B***, to the location represented by the register ***I*** or ***J***.
`cp8` ***n8***`, (%I|%J)`	Assign in memory the 8-bit constant, to the location represented by the register ***I*** or ***J***.
`cp8 (%I)`	Copy the 8-bit memory content at the address represented by register ***I***, to the memory location represented by the register ***J***.
`cp8 (%J)`	Copy the 8-bit memory content at the address represented by register ***J***, to the memory location represented by the register ***I***.
`push` ***reg***	Push at the top of the stack the 16-bit value contained inside the register.
`pop` ***reg***	Pop from the stack a 16-bit value that is then assigned to the register.
`push8` ***reg***	Push at the top of the stack the 8-bit value contained inside the register.
`pop8` ***reg***	Pop from the stack a 8-bit value that is then assigned to the register.

Macrocode syntax	Description
`push` ***n16***	Push at the top of the stack the 16-bit constant.
`push` ***n8***	Push at the top of the stack the 8-bit constant.
`push (%I|%J)`	Push at the top of the stack the 16-bit value contained in memory, at the location represented by register ***I*** or ***J***.
`push (%I|%J)`	Push at the top of the stack the 16-bit value contained in memory, at the location represented by register ***I*** or ***J***.
`push8 (%I|%J)`	Push at the top of the stack the 8-bit value contained in memory, at the location represented by register ***I*** or ***J***.
`pop (%I|%J)`	Pop from the top of the stack a 16-bit that is then copied at the memory location represented by register ***I*** or ***J***.
`pop8 (%I|%J)`	Pop from the top of the stack a 8-bit that is then copied at the memory location represented by register ***I*** or ***J***.
`jump #`***ref***	Jump to the address specified.

`jump %carry|%borrow↩`
`↪ |%zero|%negative↩`
`↪ |%overflow, ↩`
`↪ 0|1, #`***ref***
	Conditional jump. The first argument is a pseudo-register that selects the flag; the second argument means the value that the flag should have; the third argument is the address where the jump should go if the condition is met.

`equal`	***C*** = ***A***, updating the flags status
`not`	***C*** = ~***A***, updating the flags status (one's complement).
`and`	***C*** = ***A*** & ***B***, updating the flags.
`nand`	***C*** = ~(***A*** & ***B***), updating the flags.

Macrocode syntax	Description
`or`	***C*** = ***A*** | ***B***, updating the flags.
`nor`	***C*** = ~(***A*** | ***B***), updating the flags.
`xor`	***C*** = ***A*** ^ ***B***, updating the flags.
`nxor`	***C*** = ~(***A*** ^ ***B***), updating the flags.
`add`	***C*** = ***A*** + ***B***, updating the flags, 16-bit.
`sub`	***C*** = ***A*** − ***B***, updating the flags, 16-bit.
`addc`	***C*** = ***A*** + ***B*** + carry, updating the flags, 16-bit.
`subb`	***C*** = ***A*** − ***B*** − borrow, updating the flags, 16-bit.
`lshl`	***C*** = lshl(***A***), updating the flags: logic shift left, 16-bit.
`lshr`	***C*** = lshr(***A***), updating the flags: logic shift right, 16-bit.
`ashl`	***C*** = ashl(***A***), updating the flags: arithmetic shift left, 16-bit.
`ashr`	***C*** = ashr(***A***), updating the flags: arithmetic shift right, 16-bit.
`rotcl`	***C*** = rotcl(***A***), updating the flags: rotation left whith carry, 16-bit.
`rotcr`	***C*** = rotcr(***A***), updating the flags: rotation right whith carry, 16-bit.
`rotl`	***C*** = rotl(***A***), updating the flags: rotation left, 16-bit.
`rotr`	***C*** = rotr(***A***), updating the flags: rotation right, 16-bit.

Macrocode syntax	Description
`add8`	***C*** = ***A*** + ***B***, updating the flags, 8-bit.
`sub8`	***C*** = ***A*** − ***B***, updating the flags, 8-bit.
`add8c`	***C*** = ***A*** + ***B*** + carry, updating the flags, 8-bit.
`sub8b`	***C*** = ***A*** − ***B*** − borrow, updating the flags, 8-bit.
`lsh8l`	***C*** = lsh8l(***A***), updating the flags: logic shift left, 8-bit.
`lsh8r`	***C*** = lsh8r(***A***), updating the flags: logic shift right, 8-bit.
`ash8l`	***C*** = ash8l(***A***), updating the flags: arithmetic shift left, 8-bit.
`ash8r`	***C*** = ash8r(***A***), updating the flags: arithmetic shift right, 8-bit.
`rot8cl`	***C*** = rot8cl(***A***), updating the flags: rotation left whith carry, 8-bit.
`rot8cr`	***C*** = rot8cr(***A***), updating the flags: rotation right whith carry, 8-bit.
`rot8l`	***C*** = rot8l(***A***), updating the flags: rotation left, 8-bit.
`rot8r`	***C*** = rot8r(***A***), updating the flags: rotation right, 8-bit.
`cast 0|1 %A|%B`	Adapt the register value to 8-bit, unsigned or signed, depending on the first argument.
`equals`	Read the 16-bit value on top of the stack and update the flags.
`nots`	Read the 16-bit value on top of the stack and replace it with the one's complement, updating the flags.

Macrocode syntax	Description
`ands`	Pop two 16-bit values from the top of the stack and push the bitwise AND of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`nands`	Pop two 16-bit values from the top of the stack and push the bitwise NAND of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`ors`	Pop two 16-bit values from the top of the stack and push the bitwise OR of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`nors`	Pop two 16-bit values from the top of the stack and push the bitwise NOR of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`xors`	Pop two 16-bit values from the top of the stack and push the bitwise XOR of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`nxors`	Pop two 16-bit values from the top of the stack and push the bitwise NXOR of them, updating the flags, the registers ***A*** and ***B*** are overwritten.
`adds`	Pop two 16-bit values from the top of the stack and push the sum of them, updating the flags. To do the calculation, the registers ***A*** and ***B*** are overwritten.
`subs`	Pop a 16-bit values from the top of the stack and put it inside the register ***B***; then pop another 16-bit value and put inside the register ***A***; then calculate ***A*−*B*** updating the flags and push the result inside the stack again.
`addcs`	Pop two 16-bit values from the top of the stack and push the sum with carry of them, updating the flags. To do the calculation, the registers ***A*** and ***B*** are overwritten.

Macrocode syntax	Description
`subbs`	Pop a 16-bit values from the top of the stack and put it inside the register ***B***; then pop another 16-bit value and put inside the register ***A***; then calculate ***A*−*B***−borrow updating the flags and push the result inside the stack again.
`lshls`	Pop a 16-bit values from the top of the stack and push the logic shift left of it, updating the flags.
`lshrs`	Pop a 16-bit values from the top of the stack and push the logic shift right of it, updating the flags.
`ashls`	Pop a 16-bit values from the top of the stack and push the arithmetic shift left of it, updating the flags.
`ashrs`	Pop a 16-bit values from the top of the stack and push the arithmetic shift right of it, updating the flags.
`rotcls`	Pop a 16-bit values from the top of the stack and push the rotation left with carry of it, updating the flags.
`rotcrs`	Pop a 16-bit values from the top of the stack and push the rotation right with carry of it, updating the flags.
`rotls`	Pop a 16-bit values from the top of the stack and push the rotation left of it, updating the flags.
`rotrs`	Pop a 16-bit values from the top of the stack and push the rotation right of it, updating the flags.
`add8s`	Pop two 8-bit values from the top of the stack and push the sum of them, updating the flags. To do the calculation, the registers ***A*** and ***B*** are overwritten.

Macrocode syntax	Description
`sub8s`	Pop a 8-bit values from the top of the stack and put it inside the register ***B***; then pop another 8-bit value and put inside the register ***A***; then calculate ***A*−*B*** updating the flags and push the result inside the stack again.
`add8cs`	Pop two 8-bit values from the top of the stack and push the sum with carry of them, updating the flags. To do the calculation, the registers ***A*** and ***B*** are overwritten.
`sub8bs`	Pop a 8-bit values from the top of the stack and put it inside the register ***B***; then pop another 8-bit value and put inside the register ***A***; then calculate ***A*−*B***−borrow updating the flags and push the result inside the stack again.
`lsh8ls`	Pop a 8-bit values from the top of the stack and push the logic shift left of it, updating the flags.
`lsh8rs`	Pop a 8-bit values from the top of the stack and push the logic shift right of it, updating the flags.
`ash8ls`	Pop a 8-bit values from the top of the stack and push the arithmetic shift left of it, updating the flags.
`ash8rs`	Pop a 8-bit values from the top of the stack and push the arithmetic shift right of it, updating the flags.
`rotc8ls`	Pop a 8-bit values from the top of the stack and push the rotation left with carry of it, updating the flags.
`rotc8rs`	Pop a 8-bit values from the top of the stack and push the rotation right with carry of it, updating the flags.
`rot8ls`	Pop a 8-bit values from the top of the stack and push the rotation left of it, updating the flags.

Macrocode syntax	Description
`rot8rs`	Pop a 8-bit values from the top of the stack and push the rotation right of it, updating the flags.
`casts 0|1`	Pop a 16-bit value from the top of the stack and push the same value, adapted to 8-bit, unsigned or signed, depending on the argument.
`call #`***ref***	Call the procedure that starts at the argument reference: the registers ***FL*** and ***PC*** are pushed inside the stack.
`call (%I|%J)`	Call the procedure that starts at the position contained inside register ***I*** or ***J***: the registers ***FL*** and ***PC*** are pushed inside the stack.
`return`	Return from a previous call: the registers ***PC*** and ***FL*** are retrieved from the stack.
`int` ***n8***	Software interrupt call: the registers ***FL*** and ***PC*** are pushed inside the stack.
`iret`	Return from an interrupt: the registers ***PC*** and ***FL*** are retrieved from the stack.
`imrl` ***n8***	Load the IMR: interrupt mask register.
`ivtl #`***ref***	Load the IVT: interrupt vector table. The argument is the interrupt vector table start address in memory.
`flag_i 0|1`	Reset or set the flag that allow the hardware interrupts.
`flag_c 0|1`	Reset or set the carry (borrow) flag.
`in` ***n8*****`, %A|%B`**	Read from the I/O address specified, a byte that is copied inside register ***A*** or ***B***.
`in` ***n8*****`, (%I|%J)`**	Read from the I/O address specified, a byte that is copied in memory, at the address specified by register ***I*** or ***J***.

Macrocode syntax	Description
`out %A|%B,` ***n8***	Write to the I/O address specified, a byte that is read from register ***A*** or ***B***.
`out (%I|%J),` ***n8***	Write to the I/O address specified, a byte that is read from memory, at the address specified by register ***I*** or ***J***.
`ifack_jump` ***n8*****,** `#`***ref***	If the I/O port gives an acknoledge report, then jump to the specified address.
`ifack_call` ***n8*****,** `#`***ref***	If the I/O port gives an acknoledge report, then call the specified procedure.
`cmpr`	Calculate ***A−B***, updating the flags, without modifying the accumulator.
`testr`	Calculate ***A&B*** (bitwise AND), updating the flags, without modifying the accumulator.
`cmpi`	Load ***A*** with the value contained in memory at the address represented by the register ***I***; load ***B*** with the value contained in memory at the address represented by the register ***J***; then calculate ***A−B***, updating the flags, without modifying the accumulator.
`testi`	Load ***A*** with the value contained in memory at the address represented by the register ***I***; load ***B*** with the value contained in memory at the address represented by the register ***J***; then calculate ***A&B*** (bitwise AND), updating the flags, without modifying the accumulator.
`cmps`	Read from the top of the stack a 16-bit value and put it inside the register ***B***; read from the stack, just under the previous value, a 16-bit value and put it inside the register ***A***; then calculate ***A−B***, updating the flags, without modifying the accumulator.

Macrocode syntax	Description
`tests`	Read from the top of the stack a 16-bit value and put it inside the register ***B***; read from the stack, just under the previous value, a 16-bit value and put it inside the register ***A***; then calculate ***A&B***, updating the flags, without modifying the accumulator.
`cmp8r`	Calculate 8-bit ***A−B***, updating the flags, without modifying the accumulator.
`test8r`	Calculate 8-bit ***A&B*** (bitwise AND), updating the flags, without modifying the accumulator.
`cmp8i`	Load 8-bit inside ***A*** with the value contained in memory at the address represented by the register ***I***; load 8-bit inside ***B*** with the value contained in memory at the address represented by the register ***J***; then calculate ***A−B***, updating the flags, without modifying the accumulator.
`test8i`	Load 8-bit inside ***A*** with the value contained in memory at the address represented by the register ***I***; load 8-bit inside ***B*** with the value contained in memory at the address represented by the register ***J***; then calculate ***A&B***, updating the flags, without modifying the accumulator.
`cmp8s`	Read from the top of the stack a 8-bit value and put it inside the register ***B***; read from the stack, just under the previous value, a 8-bit value and put it inside the register ***A***; then calculate ***A−B***, updating the flags, without modifying the accumulator.

Macrocode syntax	Description
`test8s`	Read from the top of the stack a 8-bit value and put it inside the register ***B***; read from the stack, just under the previous value, a 8-bit value and put it inside the register ***A***; then calculate ***A*** & ***B***, updating the flags, without modifying the accumulator.
`stop`	Stop the system.

Listing 12.42. Registers, flags and opcodes encoded for TKGate. This encoding is the same as the previous version, because the opcodes do not change.

```
registers I=0, J=1, A=2, B=3, BP=4, SP=5, C=6, FL=7;

registers carry=0, borrow=0, zero=1, negative=2, overflow=3;
//-----------------------------------------------------------------------
op nop {
  map nop:         0x00;  // not operate
  operands { - = { +0=0x00; }; };
};
//
op cp {
  //                          00...... = cp %,%
  map nop:         0x00;  // 00000000 = cp %I,%I  = nop
  map cp_i_j:      0x01;  // 00000001 = cp %I,%J
  map cp_i_a:      0x02;  // 00000010 = cp %I,%A
  map cp_i_b:      0x03;  // 00000011 = cp %I,%B
  map cp_i_bp:     0x04;  // 00000100 = cp %I,%BP
  map cp_i_sp:     0x05;  // 00000101 = cp %I,%SP
  map cp_i_c:      0x06;  // 00000110 = cp %I,%C
  map cp_i_fl:     0x07;  // 00000111 = cp %I,%FL
  map cp_j_i:      0x08;  // 00001000 = cp %J,%I
  map nop:         0x09;  // 00001001 = cp %J,%J  = nop
  map cp_j_a:      0x0A;  // 00001010 = cp %J,%A
  map cp_j_b:      0x0B;  // 00001011 = cp %J,%B
  map cp_j_bp:     0x0C;  // 00001100 = cp %J,%BP
  map cp_j_sp:     0x0D;  // 00001101 = cp %J,%SP
  map cp_j_c:      0x0E;  // 00001110 = cp %J,%C
  map cp_j_fl:     0x0F;  // 00001111 = cp %J,%FL
```

```
map cp_a_i:      0x10;  // 00010000 = cp %A,%I
map cp_a_j:      0x11;  // 00010001 = cp %A,%J
map nop:         0x12;  // 00010010 = cp %A,%A  = nop
map cp_a_b:      0x13;  // 00010011 = cp %A,%B
map cp_a_bp:     0x14;  // 00010100 = cp %A,%BP
map cp_a_sp:     0x15;  // 00010101 = cp %A,%SP
map cp_a_c:      0x16;  // 00010110 = cp %A,%C
map cp_a_fl:     0x17;  // 00010111 = cp %A,%FL
map cp_b_i:      0x18;  // 00011000 = cp %B,%I
map cp_b_j:      0x19;  // 00011001 = cp %B,%J
map cp_b_a:      0x1A;  // 00011010 = cp %B,%A
map nop:         0x1B;  // 00011011 = cp %B,%B  = nop
map cp_b_bp:     0x1C;  // 00011100 = cp %B,%BP
map cp_b_sp:     0x1D;  // 00011101 = cp %B,%SP
map cp_b_c:      0x1E;  // 00011110 = cp %B,%C
map cp_b_fl:     0x1F;  // 00011111 = cp %B,%FL
map cp_bp_i:     0x20;  // 00100000 = cp %BP,%I
map cp_bp_j:     0x21;  // 00100001 = cp %BP,%J
map cp_bp_a:     0x22;  // 00100010 = cp %BP,%A
map cp_bp_b:     0x23;  // 00100011 = cp %BP,%B
map nop:         0x24;  // 00100100 = cp %BP,%BP = nop
map cp_bp_sp:    0x25;  // 00100101 = cp %BP,%SP
map cp_bp_c:     0x26;  // 00100110 = cp %BP,%C  = nop
map cp_bp_fl:    0x27;  // 00100111 = cp %BP,%FL
map cp_sp_i:     0x28;  // 00101000 = cp %SP,%I
map cp_sp_j:     0x29;  // 00101001 = cp %SP,%J
map cp_sp_a:     0x2A;  // 00101010 = cp %SP,%A
map cp_sp_b:     0x2B;  // 00101011 = cp %SP,%B
map cp_sp_bp:    0x2C;  // 00101100 = cp %SP,%BP
map nop:         0x2D;  // 00101101 = cp %SP,%SP = nop
map cp_sp_c:     0x2E;  // 00101110 = cp %SP,%C
map cp_sp_fl:    0x2F;  // 00101111 = cp %SP,%FL
map cp_c_i:      0x30;  // 00110000 = cp %C,%I
map cp_c_j:      0x31;  // 00110001 = cp %C,%J
map cp_c_a:      0x32;  // 00110010 = cp %C,%A
map cp_c_b:      0x33;  // 00110011 = cp %C,%B
map cp_c_bp:     0x34;  // 00110100 = cp %C,%BP
map cp_c_sp:     0x35;  // 00110101 = cp %C,%SP
map nop:         0x36;  // 00110110 = cp %C,%C   = nop
map cp_c_fl:     0x37;  // 00110111 = cp %C,%FL
```

```
  map cp_fl_i:     0x38;  // 00111000 = cp %FL,%I
  map cp_fl_j:     0x39;  // 00111001 = cp %FL,%J
  map cp_fl_a:     0x3A;  // 00111010 = cp %FL,%A
  map cp_fl_b:     0x3B;  // 00111011 = cp %FL,%B
  map cp_fl_bp:    0x3C;  // 00111100 = cp %FL,%BP
  map cp_fl_sp:    0x3D;  // 00111101 = cp %FL,%SP
  map cp_fl_c:     0x3E;  // 00111110 = cp %FL,%C
  map nop:         0x3F;  // 00111111 = cp %FL,%FL = nop
  //                         010000.. = cp #,%
  map cp_num_i:    0x40;  // 01000000 = cp #,%I
  map cp_num_j:    0x41;  // 01000001 = cp #,%J
  map cp_num_a:    0x42;  // 01000010 = cp #,%A
  map cp_num_b:    0x43;  // 01000011 = cp #,%B
  //                         010001.. = cp (%),%
  map cp_mi_a:     0x44;  // 01000100 = cp (%I),%A
  map cp_mi_b:     0x45;  // 01000101 = cp (%I),%B
  map cp_mj_a:     0x46;  // 01000110 = cp (%J),%A
  map cp_mj_b:     0x47;  // 01000111 = cp (%J),%B
  //                         010010.. = cp %,%I
  map cp_a_mi:     0x48;  // 01001000 = cp %A,(%I)
  map cp_a_mj:     0x49;  // 01001001 = cp %A,(%J)
  map cp_b_mi:     0x4A;  // 01001010 = cp %B,(%I)
  map cp_b_mj:     0x4B;  // 01001011 = cp %B,(%J)
  //                         0100110. = cp #,(%)
  map cp_num_mi:   0x4C;  // 01001100 = cp #,(%I)
  map cp_num_mj:   0x4D;  // 01001101 = cp #,(%J)
  //                         0100111. = cp (%)
  map cp_mi_mj:    0x4E;  // 01001110 = cp (%I)
  map cp_mj_mi:    0x4F;  // 01001111 = cp (%J)
  //
  operands {
    %1,%2   = { +0=0x00; +0[5:3]=%1; +0[2:0]=%2; };
    #1,%2   = { +0=0x40; +0[1:0]=%2; +1=#1[7:0]; +2=#1[15:8]; };
    (%1),%2 = { +0=0x44; +0[1:1]=%1; +0[0:0]=%2; };
    %1,(%2) = { +0=0x48; +0[1:1]=%1; +0[0:0]=%2; };
    #1,(%2) = { +0=0x4C; +0[0:0]=%2; +1=#1[7:0]; +2=#1[15:8]; };
    (%1)    = { +0=0x4E; +0[0:0]=%1; };
  };
};
op cp8 {
```

```
  //                          010100.. = cp8 #,%
  map cp8_num_i:  0x50;  // 01010000 = cp8 #,%I
  map cp8_num_j:  0x51;  // 01010001 = cp8 #,%J
  map cp8_num_a:  0x52;  // 01010010 = cp8 #,%A
  map cp8_num_b:  0x53;  // 01010011 = cp8 #,%B
  //                          010101.. = cp8 (%),%
  map cp8_mi_a:   0x54;  // 01010100 = cp8 (%I),%A
  map cp8_mi_b:   0x55;  // 01010101 = cp8 (%I),%B
  map cp8_mj_a:   0x56;  // 01010110 = cp8 (%J),%A
  map cp8_mj_b:   0x57;  // 01010111 = cp8 (%J),%B
  //                          010110.. = cp8 %,(%)
  map cp8_a_mi:   0x58;  // 01011000 = cp8 %A,(%I)
  map cp8_a_mj:   0x59;  // 01011001 = cp8 %A,(%J)
  map cp8_b_mi:   0x5A;  // 01011010 = cp8 %B,(%I)
  map cp8_b_mj:   0x5B;  // 01011011 = cp8 %B,(%J)
  //                          0101110. = cp8 #,(%)
  map cp8_num_mi: 0x5C;  // 01011100 = cp8 #,(%I)
  map cp8_num_mj: 0x5D;  // 01011101 = cp8 #,(%J)
  //                          0101111. = cp8 (%)
  map cp8_mi_mj:  0x5E;  // 01011110 = cp8 (%I)
  map cp8_mj_mi:  0x5F;  // 01011111 = cp8 (%J)
  //
  operands {
    #1,%2     = { +0=0x50; +0[1:0]=%2; +1=#1[7:0]; };
    (%1),%2   = { +0=0x54; +0[1:1]=%1; +0[0:0]=%2; };
    %1,(%2)   = { +0=0x58; +0[1:1]=%1; +0[0:0]=%2; };
    #1,(%2)   = { +0=0x5C; +0[0:0]=%2; +1=#1[7:0]; };
    (%1)      = { +0=0x5E; +0[0:0]=%1; };
  };
};
//
op push {
  //                          01100... = push %
  map push_i:     0x60;  // 01100000 = push %I
  map push_j:     0x61;  // 01100001 = push %J
  map push_a:     0x62;  // 01100010 = push %A
  map push_b:     0x63;  // 01100011 = push %B
  map push_bp:    0x64;  // 01100100 = push %BP
  map push_sp:    0x65;  // 01100101 = push %SP
  map push_c:     0x66;  // 01100110 = push %C
```

```
  map push_fl:     0x67;  // 01100111 = push %FL
  //
  map push_num:    0x80;  // 10000000 = push #
  //
  map push_mi:     0x82;  // 10000010 = push (%I)
  map push_mj:     0x83;  // 10000011 = push (%J)
  //
  operands {
    %1   = { +0=0x60; +0[2:0]=%1; };
    #1   = { +0=0x80; +1=#1[7:0]; +2=#1[15:8]; };
    (%1) = { +0=0x82; +0[0:0]=%1; };
  };
};
op pop {
  //                          01110... = pop %
  map pop_i:       0x70;  // 01110000 = pop %I
  map pop_j:       0x71;  // 01110001 = pop %J
  map pop_a:       0x72;  // 01110010 = pop %A
  map pop_b:       0x73;  // 01110011 = pop %B
  map pop_bp:      0x74;  // 01110100 = pop %BP
  map pop_sp:      0x75;  // 01110101 = pop %SP
  map pop_c:       0x76;  // 01110110 = pop %C
  map pop_fl:      0x77;  // 01110111 = pop %FL
  //                          1000011. = pop (%)
  map pop_mi:      0x86;  // 10000110 = pop (%I)
  map pop_mj:      0x87;  // 10000111 = pop (%J)
  //
  operands {
    %1   = { +0=0x70; +0[2:0]=%1; };
    (%1) = { +0=0x86; +0[0:0]=%1; };
  };
};
//
op push8 {
  //                          01101... = push8 %
  map push8_i:     0x68;  // 01101000 = push8 %I
  map push8_j:     0x69;  // 01101001 = push8 %J
  map push8_a:     0x6A;  // 01101010 = push8 %A
  map push8_b:     0x6B;  // 01101011 = push8 %B
  map push8_bp:    0x6C;  // 01101100 = push8 %BP
```

```
  map push8_sp:    0x6D;  // 01101101 = push8 %SP
  map push8_c:     0x6E;  // 01101110 = push8 %C
  map push8_fl:    0x6F;  // 01101111 = push8 %FL
  //
  map push8_num:   0x81;  // 10000001 = push8 #
  //                         1000010. = push8 (%I)
  map push8_mi:    0x84;  // 10000100 = push8 (%I)
  map push8_mj:    0x85;  // 10000101 = push8 (%J)
  //
  operands {
    %1   = { +0=0x68; +0[2:0]=%1; };
    #1   = { +0=0x81; +1[7:0]=#1; };
    (%1) = { +0=0x81; +0[0:0]=%1; };
  };
};
op pop8 {
  //                         01111... = pop8 %
  map pop8_i:      0x78;  // 01111000 = pop8 %I
  map pop8_j:      0x79;  // 01111001 = pop8 %J
  map pop8_a:      0x7A;  // 01111010 = pop8 %A
  map pop8_b:      0x7B;  // 01111011 = pop8 %B
  map pop8_bp:     0x7C;  // 01111100 = pop8 %BP
  map pop8_sp:     0x7D;  // 01111101 = pop8 %SP
  map pop8_c:      0x7E;  // 01111110 = pop8 %C
  map pop8_fl:     0x7F;  // 01111111 = pop8 %FL
  //                       // 1000100. = pop8 (%)
  map pop8_mi:     0x88;  // 10001000 = pop8 (%I)
  map pop8_mj:     0x89;  // 10001001 = pop8 (%J)
  //
  operands {
    %1   = { +0=0x78; +0[2:0]=%1; };
    (%1) = { +0=0x88; +0[0:0]=%1; };
  };
};
//
op jump {
  map jump_nc:     0x90;  // 10010000 = jump %carry 0 #
  map jump_c:      0x91;  // 10010001 = jump %carry 1 #
  map jump_nz:     0x92;  // 10010010 = jump %zero  0 #
  map jump_z:      0x93;  // 10010011 = jump %zero  1 #
```

```
  map jump_nn:     0x94;  // 10010100 = jump %negative 0 #
  map jump_n:      0x95;  // 10010101 = jump %negative 1 #
  map jump_no:     0x96;  // 10010110 = jump %overflow 0 #
  map jump_o:      0x97;  // 10010111 = jump %overflow 1 #
  //
  map jump:        0xFE;  // 11111110 = jump #
  //
  operands {
     %1,#2,#3 = { +0=0x90; +0[2:1]=%1; +0[0:0]=#2;
                  +1=#3[7:0]; +2=#3[15:8]; };
     #1       = { +0=0xFE; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
//
op equal {
  map equal:       0x98;  // 10011000 = equal
  operands { - = { +0=0x98; }; };
};
op not {
  map not:         0x99;  // 10011001 = not
  operands { - = { +0=0x99; }; };
};
//
op and {
  map and:         0x9A;  // 10011010 = and
  operands { - = { +0=0x9A; }; };
};
op nand {
  map nand:        0x9B;  // 10011011 = nand
  operands { - = { +0=0x9B; }; };
};
//
op or {
  map or:          0x9C;  // 10011100 = or
  operands { - = { +0=0x9C; }; };
};
op nor {
  map nor:         0x9D;  // 10011101 = nor
  operands { - = { +0=0x9D; }; };
};
```

```
//
op xor {
  map xor:          0x9E;  // 10011110 = xor
  operands { - = { +0=0x9E; }; };
};
op nxor {
  map nxor:         0x9F;  // 10011111 = nxor
  operands { - = { +0=0x9F; }; };
};
//
op add {
  map add:          0xA0;  // 10100000 = add
  operands { - = { +0=0xA0; }; };
};
op sub {
  map sub:          0xA1;  // 10100001 = sub
  operands { - = { +0=0xA1; }; };
};
//
op addc {
  map addc:         0xA2;  // 10100010 = addc
  operands { - = { +0=0xA2; }; };
};
op subb {
  map subb:         0xA3;  // 10100011 = subb
  operands { - = { +0=0xA3; }; };
};
//
op lshl {
  map lshl:         0xA4;  // 10100100 = lshl
  operands { - = { +0=0xA4; }; };
};
op lshr {
  map lshr:         0xA5;  // 10100101 = lshr
  operands { - = { +0=0xA5; }; };
};
//
op ashl {
  map ashl:         0xA6;  // 10100110 = ashl
  operands { - = { +0=0xA6; }; };
```

```
};
op ashr {
  map ashr:        0xA7;  // 10100111 = ashr
  operands { - = { +0=0xA7; }; };
};
//
op rotcl {
  map rotcl:       0xA8;  // 10101000 = rotcl
  operands { - = { +0=0xA8; }; };
};
op rotcr {
  map rotcr:       0xA9;  // 10101001 = rotcr
  operands { - = { +0=0xA9; }; };
};
//
op rotl {
  map rotl:        0xAA;  // 10101010 = rotl
  operands { - = { +0=0xAA; }; };
};
op rotr {
  map rotr:        0xAB;  // 10101011 = rotr
  operands { - = { +0=0xAB; }; };
};
//
op add8 {
  map add8:        0xAC;  // 10101100 = add8
  operands { - = { +0=0xAC; }; };
};
op sub8 {
  map sub8:        0xAD;  // 10101101 = sub8
  operands { - = { +0=0xAD; }; };
};
//
op add8c {
  map add8c:       0xAE;  // 10101110 = addc8
  operands { - = { +0=0xAE; }; };
};
op sub8b {
  map sub8b:       0xAF;  // 10101111 = subb8
  operands { - = { +0=0xAF; }; };
```

```
};
//
op lsh8l {
  map lsh8l:      0xB0;  // 10110000 = lsh18l
  operands { - = { +0=0xB0; }; };
};
op lsh8r {
  map lsh8r:      0xB1;  // 10110001 = lsh18r
  operands { - = { +0=0xB1; }; };
};
//
op ash8l {
  map ash8l:      0xB2;  // 10110010 = ash18l
  operands { - = { +0=0xB2; }; };
};
op ash8r {
  map ash8r:      0xB3;  // 10110011 = ash18r
  operands { - = { +0=0xB3; }; };
};
//
op rot8cl {
  map rot8cl:     0xB4;  // 10110100 = rotc81
  operands { - = { +0=0xB4; }; };
};
op rot8cr {
  map rot8cr:     0xB5;  // 10110101 = rotc8r
  operands { - = { +0=0xB5; }; };
};
//
op rot8l {
  map rot8l:      0xB6;  // 10110110 = rot81
  operands { - = { +0=0xB6; }; };
};
op rot8r {
  map rot8r:      0xB7;  // 10110111 = rot8r
  operands { - = { +0=0xB7; }; };
};
//
op cast {
  //                    // 101110.. = cast <sig>,%
```

```
  map cast_uns_a: 0xB8;  // 10111000 = cast 0,%A
  map cast_uns_b: 0xB9;  // 10111001 = cast 0,%B
  map cast_sig_a: 0xBA;  // 10111010 = cast 1,%A
  map cast_sig_b: 0xBB;  // 10111011 = cast 1,%B
  operands {
    #1,%2 = {+0=0xB8; +0[1:1]=#1; +0[0:0]=%2; };
  };
};
//
op equals {
  map equals:     0xBC;  // 10111100 = stack <= %A
  operands { - = { +0=0xBC; }; };
};
op nots {
  map nots:       0xBD;  // 10111101 = stack <= NOT %A
  operands { - = { +0=0xBD; }; };
};
//
op ands {
  map ands:       0xBE;  // 10111110 = stack <= %A AND %B
  operands { - = { +0=0xBE; }; };
};
op nands {
  map nands:      0xBF;  // 10111111 = stack <= %A NAND %B
  operands { - = { +0=0xBF; }; };
};
//
op ors {
  map ors:        0xC0;  // 11000000 = stack <= %A OR %B
  operands { - = { +0=0xC0; }; };
};
op nors {
  map nors:       0xC1;  // 11000001 = stack <= %A NOR %B
  operands { - = { +0=0xC1; }; };
};
//
op xors {
  map xors:       0xC2;  // 11000010 = stack <= %A XOR %B
  operands { - = { +0=0xC2; }; };
};
```

```
op nxors {
  map nxors:       0xC3;  // 11000011 = stack <= %A NXOR %B
  operands { - = { +0=0xC3; }; };
};
//
op adds {
  map adds:        0xC4;  // 11000100 = stack <= %A + %B
  operands { - = { +0=0xC4; }; };
};
op subs {
  map subs:        0xC5;  // 11000101 = stack <= %A - %B
  operands { - = { +0=0xC5; }; };
};
//
op addcs {
  map addcs:       0xC6;  // 11000110 = stack <= %A + %B + %carry
  operands { - = { +0=0xC6; }; };
};
op subbs {
  map subbs:       0xC7;  // 11000111 = stack <= %A - %B - %carry
  operands { - = { +0=0xC7; }; };
};
//
op lshls {
  map lshls:       0xC8;  // 11001000 = stack <= lshl
  operands { - = { +0=0xC8; }; };
};
op lshrs {
  map lshrs:       0xC9;  // 11001001 = stack <= lshr
  operands { - = { +0=0xC9; }; };
};
//
op ashls {
  map ashls:       0xCA;  // 11001010 = stack <= ashl
  operands { - = { +0=0xCA; }; };
};
op ashrs {
  map ashrs:       0xCB;  // 11001011 = stack <= ashr
  operands { - = { +0=0xCB; }; };
};
```

```
//
op rotcls {
  map rotcls:      0xCC;  // 11001100 = stack <= rotcl
  operands { - = { +0=0xCC; }; };
};
op rotcrs {
  map rotcrs:      0xCD;  // 11001101 = stack <= rotcr
  operands { - = { +0=0xCD; }; };
};
//
op rotls {
  map rotls:       0xCE;  // 11001110 = stack <= rotl
  operands { - = { +0=0xCE; }; };
};
op rotrs {
  map rotrs:       0xCF;  // 11001111 = stack <= rotr
  operands { - = { +0=0xCF; }; };
};
//
op add8s {
  map add8s:       0xD0;  // 11010000 = stack8 <= +
  operands { - = { +0=0xD0; }; };
};
op sub8s {
  map sub8s:       0xD1;  // 11010001 = stack8 <= -
  operands { - = { +0=0xD1; }; };
};
//
op add8cs {
  map add8cs:      0xD2;  // 11010010 = stack8 <= + + %carry
  operands { - = { +0=0xD2; }; };
};
op sub8bs {
  map sub8bs:      0xD3;  // 11010011 = stack8 <= - - %carry
  operands { - = { +0=0xD3; }; };
};
//
op lsh8ls {
  map lsh8ls:      0xD4;  // 11010100 = stack8 <= lsh8r
  operands { - = { +0=0xD4; }; };
```

```
};
op lsh8rs {
  map lsh8rs:     0xD5;  // 11010101 = stack8 <= lsh8r
  operands { - = { +0=0xD5; }; };
};
//
op ash8ls {
  map ash8ls:     0xD6;  // 11010110 = stack8 <= ash8l
  operands { - = { +0=0xD6; }; };
};
op ash8rs {
  map ash8rs:     0xD7;  // 11010111 = stack8 <= ash8r
  operands { - = { +0=0xD7; }; };
};
//
op rot8cls {
  map rot8cls:    0xD8;  // 11011000 = stack8 <= rot8cl
  operands { - = { +0=0xD8; }; };
};
op rot8crs {
  map rot8crs:    0xD9;  // 11011001 = stack8 <= rot8cr
  operands { - = { +0=0xD9; }; };
};
//
op rot8ls {
  map rot8ls:     0xDA;  // 11011010 = stack8 <= rot8l
  operands { - = { +0=0xDA; }; };
};
op rot8rs {
  map rot8rs:     0xDB;  // 11011011 = stack8 <= rot8r
  operands { - = { +0=0xDB; }; };
};
//
op casts {
  //                    // 1101110. = casts <sig>
  map casts_uns:  0xDC;  // 11011100 = casts 0
  map casts_sig:  0xDD;  // 11011101 = casts 1
  operands {
    #1 = {+0=0xDC; +0[0:0]=#1; };
  };
```

```
};
//
op call {
  map call:        0xDE;  // 11011110 = call #
  map call_i:      0xFC;  // 11111100 = call (%I)
  map call_j:      0xFD;  // 11111101 = call (%J)
  operands {
    #1   = { +0=0xDE; +1=#1[7:0]; +2=#1[15:8]; };
    (%1) = { +0=0xFC; +0[0:0]=%1; };
  };
};
op return {
  map return:      0xDF;  // 11011111 = return
  operands { - = { +0=0xDF; }; };
};
//
op int {
  map int:         0xE0;  // 11100000 = int <n>
  operands {
    #1 = { +0=0xE0; +1=#1; };
  };
};
op iret {
  map iret:        0xE1;  // 11100001 = iret
  operands { - = { +0=0xE1; }; };
};
//
op imrl {
  map imrl:        0xE2;  // 11100010 = IMR load
  operands {
    #1   = { +0=0xE2; +1=#1[7:0]; };
  };
  operands { - = { +0=0xE2; }; };
};
op ivtl {
  map ivtl:        0xE3;  // 11100011 = IVT load
  operands {
    #1   = { +0=0xE3; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
```

```
//
op flag_i {
  map flag_i_clr: 0xE4;  // 11100100 = flag_i 0
  map flag_i_set: 0xE5;  // 11100101 = flag_i 1
  operands {
    #1 = { +0=0xE4; +0[0:0]=#1; };
  };
};
//
op flag_c {
  map flag_c_clr: 0xE6;  // 11100110 = flag_c 0
  map flag_c_set: 0xE7;  // 11100111 = flag_c 1
  operands {
    #1 = { +0=0xE6; +0[0:0]=#1; };
  };
};
//
op in {
  map in_num_a:   0xE8;  // 11101000 = in <n>,%A
  map in_num_b:   0xE9;  // 11101001 = in <n>,%B
  map in_num_mi:  0xEA;  // 11101010 = in <n>,(%I)
  map in_num_mj:  0xEB;  // 11101011 = in <n>,(%J)
  operands {
    #1,%2   = { +0=0xE8; +0[0:0]=%2; +1=#1; };
    #1,(%2) = { +0=0xEA; +0[0:0]=%2; +1=#1; };
  };
};
op out {
  map out_a_num:  0xEC;  // 11101100 = out %A,#
  map out_b_num:  0xED;  // 11101101 = out %B,#
  map out_mi_num: 0xEE;  // 11101110 = out (%I),#
  map out_mi_num: 0xEF;  // 11101111 = out (%J),#
  operands {
    %1,#2   = { +0=0xEC; +0[0:0]=%1; +1=#2; };
    (%1),#2 = { +0=0xEE; +0[0:0]=%1; +1=#2; };
  };
};
//
op ifack_jump {
  map ifack_jump: 0xF0;  // 11110000 = jump if ack from device
```

```
  operands {
    #1,#2    = { +0=0xF0; +1=#1; +2=#2[7:0]; +3=#2[15:8]; };
  };
};
op ifack_call {
  map ifack_call: 0xF1;  // 11110001 = call if ack from device
  operands {
    #1,#2    = { +0=0xF1; +1=#1; +2=#2[7:0]; +3=#2[15:8]; };
  };
};
//
op cmpr {
  map cmpr:        0xF2;  // 11110010 = compare %A, %B
  operands { - = { +0[7:0]=0xF2; }; };
};
op testr {
  map testr:       0xF3;  // 11110011 = test %A, %B
  operands { - = { +0[7:0]=0xF3; }; };
};
//
op cmpi {
  map cmpi:        0xF4;  // 11110100 = compare (%I), (%J)
  operands { - = { +0[7:0]=0xF4; }; };
};
op testi {
  map testi:       0xF5;  // 11110101 = test (%I), (%J)
  operands { - = { +0[7:0]=0xF5; }; };
};
//
op cmps {
  map cmps:        0xF6;  // 11110110 = compare stack
  operands { - = { +0[7:0]=0xF6; }; };
};
op tests {
  map tests:       0xF7;  // 11110111 = test stack
  operands { - = { +0[7:0]=0xF7; }; };
};
//
op cmp8i {
  map cmp8i:       0xF8;  // 11111000 = compare8 (%I), (%J)
```

```
  operands { - = { +0[7:0]=0xF8; }; };
};
op test8i {
  map test8i:     0xF9;  // 11111001 = test8 (%I), (%J)
  operands { - = { +0[7:0]=0xF9; }; };
};
//
op cmp8s {
  map cmp8s:      0xFA;  // 11111010 = compare8 stack
  operands { - = { +0[7:0]=0xFA; }; };
};
op test8s {
  map test8s:     0xFB;  // 11111011 = test8 stack
  operands { - = { +0[7:0]=0xFB; }; };
};
//
op stop {
  map stop:       0xFF;  // 11111111
  operands { - = { +0[7:0]=0xFF; }; };
};
```

12.14 Microcode

«

It follows the microcode listing, describing the opcode procedures. The microcode is much different from the previous version, because the bit fields have different names and because the memory is accessed in a new slower way. Please notice that the fetch cycle is started with just **`fetch=f1`**, but it requires two clock cycles and a separate **`ctrl=load`** is also required.

Listing 12.43. Microcode for TKGate.

```
begin microcode @ 0
//
fetch1:
    // MAR <= PC++, MIR <= RAM  == fetch=f1;
    mar=p mar=rank8 mar=load pc=dp1 mir=ram_read;
fetch0:
    // MIR <= RAM, MPC <= MIR
    ctrl=load;
//
nop:
```

```
    // MPC <= #fetch
    addr=fetch1 jump=true;
//
cp_i_j:
    fetch=f1 j=a j=load i=a i=bw;          // J <= I
    ctrl=load;
cp_i_a:
    fetch=f1 a=a a=load i=a i=bw;          // A <= I
    ctrl=load;
cp_i_b:
    fetch=f1 b=a b=load i=a i=bw;          // B <= I
    ctrl=load;
cp_i_sp:
    fetch=f1 sp=a sp=load i=a i=bw;        // SP <= I
    ctrl=load;
cp_i_bp:
    fetch=f1 bp=a bp=load i=a i=bw;        // BP <= I
    ctrl=load;
cp_i_c:
    fetch=f1 c=a c=load i=a i=bw;          // C <= I
    ctrl=load;
cp_i_fl:
    fetch=f1 fl=a fl=load i=a i=bw;        // FL <= I
    ctrl=load;
cp_j_i:
    fetch=f1 i=a i=load j=a j=bw;          // I <= J
    ctrl=load;
cp_j_a:
    fetch=f1 a=a a=load j=a j=bw;          // A <= J
    ctrl=load;
cp_j_b:
    fetch=f1 b=a b=load j=a j=bw;          // B <= J
    ctrl=load;
cp_j_sp:
    fetch=f1 sp=a sp=load j=a j=bw;        // SP <= J
    ctrl=load;
cp_j_bp:
    fetch=f1 bp=a bp=load j=a j=bw;        // BP <= J
    ctrl=load;
cp_j_c:
    fetch=f1 c=a c=load j=a j=bw;          // C <= J
    ctrl=load;
cp_j_fl:
```

```
    fetch=f1 fl=a fl=load j=a j=bw;      // FL <= J
    ctrl=load;
cp_a_i:
    fetch=f1 i=a i=load a=a a=bw;        // I <= A
    ctrl=load;
cp_a_j:
    fetch=f1 j=a j=load a=a a=bw;        // J <= A
    ctrl=load;
cp_a_b:
    fetch=f1 b=a b=load a=a a=bw;        // B <= A
    ctrl=load;
cp_a_sp:
    fetch=f1 sp=a sp=load a=a a=bw;      // SP <= A
    ctrl=load;
cp_a_bp:
    fetch=f1 bp=a bp=load a=a a=bw;      // BP <= A
    ctrl=load;
cp_a_c:
    fetch=f1 c=a c=load a=a a=bw;        // C <= A
    ctrl=load;
cp_a_fl:
    fetch=f1 fl=a fl=load a=a a=bw;      // FL <= A
    ctrl=load;
cp_b_i:
    fetch=f1 i=a i=load b=a b=bw;        // I <= B
    ctrl=load;
cp_b_j:
    fetch=f1 j=a j=load b=a b=bw;        // J <= B
    ctrl=load;
cp_b_a:
    fetch=f1 a=a a=load b=a b=bw;        // A <= B
    ctrl=load;
cp_b_sp:
    fetch=f1 sp=a sp=load b=a b=bw;      // SP <= B
    ctrl=load;
cp_b_bp:
    fetch=f1 bp=a bp=load b=a b=bw;      // BP <= B
    ctrl=load;
cp_b_c:
    fetch=f1 c=a c=load b=a b=bw;        // C <= B
    ctrl=load;
cp_b_fl:
    fetch=f1 fl=a fl=load b=a b=bw;      // FL <= B
```

```
    ctrl=load;
cp_sp_i:
    fetch=f1 i=a i=load sp=a sp=bw;      // I <= SP
    ctrl=load;
cp_sp_j:
    fetch=f1 j=a j=load sp=a sp=bw;      // J <= SP
    ctrl=load;
cp_sp_a:
    fetch=f1 a=a a=load sp=a sp=bw;      // A <= SP
    ctrl=load;
cp_sp_b:
    fetch=f1 b=a b=load sp=a sp=bw;      // B <= SP
    ctrl=load;
cp_sp_bp:
    fetch=f1 bp=a bp=load sp=a sp=bw;    // BP <= SP
    ctrl=load;
cp_sp_c:
    fetch=f1 c=a c=load sp=a sp=bw;      // C <= SP
    ctrl=load;
cp_sp_fl:
    fetch=f1 fl=a fl=load sp=a sp=bw;    // FL <= SP
    ctrl=load;
cp_bp_i:
    fetch=f1 i=a i=load bp=a bp=bw;      // I <= BP
    ctrl=load;
cp_bp_j:
    fetch=f1 j=a j=load bp=a bp=bw;      // J <= BP
    ctrl=load;
cp_bp_a:
    fetch=f1 a=a a=load bp=a bp=bw;      // A <= BP
    ctrl=load;
cp_bp_b:
    fetch=f1 b=a b=load bp=a bp=bw;      // B <= BP
    ctrl=load;
cp_bp_sp:
    fetch=f1 sp=a sp=load bp=a bp=bw;    // SP <= BP
    ctrl=load;
cp_bp_c:
    fetch=f1 c=a c=load bp=a bp=bw;      // C <= BP
    ctrl=load;
cp_bp_fl:
    fetch=f1 fl=a fl=load bp=a bp=bw;    // FL <= BP
    ctrl=load;
```

```
cp_c_i:
    fetch=f1 i=a i=load c=a c=bw;        // I <= C
    ctrl=load;
cp_c_j:
    fetch=f1 j=a j=load c=a c=bw;        // J <= C
    ctrl=load;
cp_c_a:
    fetch=f1 a=a a=load c=a c=bw;        // A <= C
    ctrl=load;
cp_c_b:
    fetch=f1 b=a b=load c=a c=bw;        // B <= C
    ctrl=load;
cp_c_sp:
    fetch=f1 sp=a sp=load c=a c=bw;      // SP <= C
    ctrl=load;
cp_c_bp:
    fetch=f1 bp=a bp=load c=a c=bw;      // BP <= C
    ctrl=load;
cp_c_fl:
    fetch=f1 fl=a fl=load c=a c=bw;      // FL <= C
    ctrl=load;
cp_fl_i:
    fetch=f1 i=a i=load fl=a fl=bw;      // I <= FL
    ctrl=load;
cp_fl_j:
    fetch=f1 j=a j=load fl=a fl=bw;      // J <= FL
    ctrl=load;
cp_fl_a:
    fetch=f1 a=a a=load fl=a fl=bw;      // A <= FL
    ctrl=load;
cp_fl_b:
    fetch=f1 b=a b=load fl=a fl=bw;      // B <= FL
    ctrl=load;
cp_fl_sp:
    fetch=f1 sp=a sp=load fl=a fl=bw;    // SP <= FL
    ctrl=load;
cp_fl_bp:
    fetch=f1 bp=a bp=load fl=a fl=bw;    // BP <= FL
    ctrl=load;
cp_fl_c:
    fetch=f1 c=a c=load fl=a fl=bw;      // C <= FL
    ctrl=load;
cp_num_i:
```

```
    // MAR <= PC+=2, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // I <= MDR;
    fetch=f1 i=a i=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
cp_num_j:
    // MAR <= PC+=2, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // J <= MDR;
    fetch=f1 j=a j=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
cp_num_a:
    // MAR <= PC+=2, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // A <= MDR;
    fetch=f1 a=a a=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
cp_num_b:
    // MAR <= PC+=2, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // B <= MDR;
    fetch=f1 b=a b=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
cp_mi_a:
    // MAR <= I, MDR <= RAM (3 clocks);
    mar=i mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // A <= MDR;
    fetch=f1 a=a a=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
cp_mi_b:
    // MAR <= I, MDR <= RAM (3 clocks);
    mar=i mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // B <= MDR;
    fetch=f1 b=a b=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
cp_mj_a:
    // MAR <= J, MDR <= RAM (3 clocks);
    mar=j mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // A <= MDR;
    fetch=f1 a=a a=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
cp_mj_b:
    // MAR <= J, MDR <= RAM (3 clocks);
```

```
    mar=j mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // B <= MDR;
    fetch=f1 b=a b=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
cp_a_mi:
    // MDR <= A
    mdr=a mdr=load a=bw;
    // MAR <= I, RAM <= MDR (3 clocks)
    mar=i mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp_a_mj:
    // MDR <= A
    mdr=a mdr=load a=bw;
    // MAR <= J, RAM <= MDR (3 clocks)
    mar=j mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp_b_mi:
    // MDR <= B
    mdr=a mdr=load b=bw;
    // MAR <= I, RAM <= MDR (3 clocks)
    mar=i mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp_b_mj:
    // MDR <= B
    mdr=a mdr=load b=bw;
    // MAR <= J, RAM <= MDR (3 clocks)
    mar=j mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp_num_mi:
    // MAR <= PC+=2, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // MAR <= I, RAM <= MDR (3 clocks)
    mar=i mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp_num_mj:
    // MAR <= PC+=2, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // MAR <= J, RAM <= MDR (3 clocks)
    mar=j mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp_mi_mj:
    // MAR <= I, MDR <= RAM (3 clocks);
    mar=i mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
```

```
    // MAR <= J, RAM <= MDR (3 clocks)
    mar=j mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp_mj_mi:
    // MAR <= J, MDR <= RAM (3 clocks);
    mar=j mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // MAR <= I, RAM <= MDR (3 clocks)
    mar=i mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp8_num_i:
    // MAR <= PC++, MDR <= RAM (2 clocks);
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // I <= MDR;
    fetch=f1 i=a i=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
cp8_num_j:
    // MAR <= PC++, MDR <= RAM (2 clocks);
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // J <= MDR;
    fetch=f1 j=a j=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
cp8_num_a:
    // MAR <= PC++, MDR <= RAM (2 clocks);
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // A <= MDR;
    fetch=f1 a=a a=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
cp8_num_b:
    // MAR <= PC++, MDR <= RAM (2 clocks);
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // B <= MDR;
    fetch=f1 b=a a=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
cp8_mi_a:
    // MAR <= I, MDR <= RAM (2 clocks);
    mar=i mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // A <= MDR;
    fetch=f1 a=a a=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
cp8_mi_b:
    // MAR <= I, MDR <= RAM (2 clocks);
    mar=i mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // B <= MDR;
```

```
    fetch=f1 b=a a=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
cp8_mj_a:
    // MAR <= J, MDR <= RAM (2 clocks);
    mar=j mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // A <= MDR;
    fetch=f1 a=a a=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
cp8_mj_b:
    // MAR <= J, MDR <= RAM (2 clocks);
    mar=j mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // B <= MDR;
    fetch=f1 b=a a=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
cp8_a_mi:
    // MDR <= A
    mdr=a mdr=load a=bw;
    // MAR <= I, RAM <= MDR (3 clocks)
    mar=i mar=rank8 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp8_a_mj:
    // MDR <= A
    mdr=a mdr=load a=bw;
    // MAR <= J, RAM <= MDR (3 clocks)
    mar=j mar=rank8 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp8_b_mi:
    // MDR <= B
    mdr=a mdr=load b=bw;
    // MAR <= I, RAM <= MDR (3 clocks)
    mar=i mar=rank8 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp8_b_mj:
    // MDR <= A
    mdr=a mdr=load b=bw;
    // MAR <= J, RAM <= MDR (3 clocks)
    mar=j mar=rank8 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
cp8_num_mi:
    // MAR <= PC++, MDR <= RAM (2 clocks);
    mar=p mar=rank8 mar=load pc=dp2 mdr=ram_read; ctrl=nop;
    // MAR <= I, RAM <= MDR (2 clocks)
    mar=i mar=rank8 mar=load mar=ram_write; ctrl=nop;
```

```
    fetch=f1; ctrl=load;
cp8_num_mj:
    // MAR <= PC++, MDR <= RAM (2 clocks);
    mar=p mar=rank8 mar=load pc=dp2 mdr=ram_read; ctrl=nop;
    // MAR <= J, RAM <= MDR (2 clocks)
    mar=j mar=rank8 mar=load mar=ram_write; ctrl=nop;
    fetch=f1; ctrl=load;
cp8_mi_mj:
    // MAR <= I, MDR <= RAM (2 clocks);
    mar=i mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // MAR <= J, RAM <= MDR (2 clocks)
    mar=j mar=rank8 mar=load mar=ram_write; ctrl=nop;
    fetch=f1; ctrl=load;
cp8_mj_mi:
    // MAR <= J, MDR <= RAM (2 clocks);
    mar=j mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // MAR <= I, RAM <= MDR (2 clocks)
    mar=i mar=rank8 mar=load mar=ram_write; ctrl=nop;
    fetch=f1; ctrl=load;
push_i:
    // MDR <= I
    mdr=a mdr=load i=a i=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push_j:
    // MDR <= J
    mdr=a mdr=load j=a j=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push_a:
    // MDR <= A
    mdr=a mdr=load a=a a=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push_b:
    // MDR <= B
    mdr=a mdr=load b=a b=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
```

```
push_bp:
    // MDR <= BP
    mdr=a mdr=load bp=a bp=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push_sp:
    // MDR <= SP
    mdr=a mdr=load sp=a sp=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push_c:
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push_fl:
    // MDR <= FL
    mdr=a mdr=load fl=a fl=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push_num:
    // MAR <= PC++, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push_mi:
    // MAR <= I, MDR <= RAM (3 clocks);
    mar=i mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push_mj:
    // MAR <= J, MDR <= RAM (3 clocks);
    mar=j mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
pop_i:
```

```
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // I <= MDR;
    fetch=f1 i=a i=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
pop_j:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // J <= MDR;
    fetch=f1 j=a j=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
pop_a:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    fetch=f1 a=a a=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
pop_b:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    fetch=f1 b=a b=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
pop_bp:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // BP <= MDR;
    fetch=f1 bp=a bp=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
pop_sp:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // SP <= MDR;
    fetch=f1 sp=a sp=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
pop_c:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // C <= MDR;
    fetch=f1 c=a c=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
pop_fl:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
```

```
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // FL <= MDR;
    fetch=f1 fl=a fl=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
pop_mi:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // MAR <= I, RAM <= MDR (3 clocks)
    mar=i mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
pop_mj:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // MAR <= J, RAM <= MDR (3 clocks)
    mar=j mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
push8_i:
    // MDR <= I
    mdr=a mdr=load i=a i=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_j:
    // MDR <= J
    mdr=a mdr=load j=a j=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_a:
    // MDR <= A
    mdr=a mdr=load a=a a=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_b:
    // MDR <= B
    mdr=a mdr=load b=a b=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_bp:
    // MDR <= BP
    mdr=a mdr=load bp=a bp=bw;
```

```
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_sp:
    // MDR <= SP
    mdr=a mdr=load sp=a sp=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_c:
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_fl:
    // MDR <= FL
    mdr=a mdr=load fl=a fl=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_num:
    // MAR <= PC++, MDR <= RAM (2 clocks);
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_mi:
    // MAR <= I, MDR <= RAM (2 clocks);
    mar=i mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
push8_mj:
    // MAR <= J, MDR <= RAM (2 clocks);
    mar=j mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
pop8_i:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // I <= MDR;
```

```
    fetch=f1 i=a i=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
pop8_j:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // J <= MDR;
    fetch=f1 j=a j=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
pop8_a:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // A <= MDR;
    fetch=f1 a=a a=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
pop8_b:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // B <= MDR;
    fetch=f1 b=a b=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
pop8_bp:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // BP <= MDR;
    fetch=f1 bp=a bp=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
pop8_sp:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // SP <= MDR;
    fetch=f1 sp=a sp=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
pop8_c:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // C <= MDR;
    fetch=f1 c=a c=load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
pop8_fl:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // FL <= MDR;
    fetch=f1 fl=a fl=load mdr=a mdr=rank8u mdr=bw;
```

```
    ctrl=load;
pop8_mi:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // MAR <= I, RAM <= MDR (2 clocks)
    mar=i mar=rank8 mar=load mar=ram_write; ctrl=nop;
    fetch=f1; ctrl=load;
pop8_mj:
    // MAR <= SP++, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // MAR <= J, RAM <= MDR (2 clocks)
    mar=j mar=rank8 mar=load mar=ram_write; ctrl=nop;
    fetch=f1; ctrl=load;
jump:
    // MAR <= PC+=2, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // PC <= MDR;
    pc=a pc=load mdr=a mdr=rank16 mdr=bw;
    fetch=f1; ctrl=load;
jump_nc:
    // check flag before reading the destination argument
    addr=jump jump=carry_f;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
jump_c:
    // check flag before reading the destination argument
    addr=jump jump=carry_t;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
jump_nz:
    // check flag before reading the destination argument
    addr=jump jump=zero_f;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
jump_z:
    // check flag before reading the destination argument
    addr=jump jump=zero_t;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
```

```
jump_nn:
    // check flag before reading the destination argument
    addr=jump jump=negative_f;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
jump_n:
    // check flag before reading the destination argument
    addr=jump jump=negative_t;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
jump_no:
    // check flag before reading the destination argument
    addr=jump jump=overflow_f;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
jump_o:
    // check flag before reading the destination argument
    addr=jump jump=overflow_t;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
equal:
    // C <= A (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=a alu=rank16 a=a a=bw;
    ctrl=load;
not:
    // C <= ~A (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=not alu=rank16 a=a a=bw;
    ctrl=load;
and:
    // C <= A&B (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=and alu=rank16 a=a a=bw b=b b=bw;
    ctrl=load;
nand:
    // C <= ~(A&B) (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=nand alu=rank16 a=a a=bw b=b b=bw;
    ctrl=load;
or:
    // C <= A|B (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=or alu=rank16 a=a a=bw b=b b=bw;
```

```
    ctrl=load;
nor:
    // C <= ~(A|B)  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=nor alu=rank16 a=a a=bw b=b b=bw;
    ctrl=load;
xor:
    // C <= A^B  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=xor alu=rank16 a=a a=bw b=b b=bw;
    ctrl=load;
nxor:
    // C <= ~(A^B)  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=nxor alu=rank16 a=a a=bw b=b b=bw;
    ctrl=load;
add:
    // C <= A+B  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=add alu=rank16 a=a a=bw b=b b=bw;
    ctrl=load;
sub:
    // C <= A-B  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=sub alu=rank16 a=a a=bw b=b b=bw;
    ctrl=load;
addc:
    // C <= A+B+carry  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=addc alu=rank16 a=a a=bw b=b b=bw;
    ctrl=load;
subb:
    // C <= A-B-borrow  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=subb alu=rank16 a=a a=bw b=b b=bw;
    ctrl=load;
lshl:
    // C <= lshl(A)  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=lshl alu=rank16 a=a a=bw;
    ctrl=load;
lshr:
    // C <= lshr(A)  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=lshr alu=rank16 a=a a=bw;
    ctrl=load;
ashl:
    // C <= ashl(A)  (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=ashl alu=rank16 a=a a=bw;
    ctrl=load;
ashr:
    // C <= ashr(A)  (FL updated)
```

```
    fetch=f1 c=b c=load fl=b fl=load alu=ashr alu=rank16 a=a a=bw;
    ctrl=load;
rotcl:
    // C <= rotcl(A) (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=rotcl alu=rank16 a=a a=bw;
    ctrl=load;
rotcr:
    // C <= rotcr(A) (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=rotcr alu=rank16 a=a a=bw;
    ctrl=load;
rotl:
    // C <= rotl(A) (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=rotl alu=rank16 a=a a=bw;
    ctrl=load;
rotr:
    // C <= rotr(A) (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=rotr alu=rank16 a=a a=bw;
    ctrl=load;
add8:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=add8 alu=rank8u a=a a=bw b=b b=bw;
    ctrl=load;
sub8:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=sub8 alu=rank8u a=a a=bw b=b b=bw;
    ctrl=load;
add8c:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=add8c alu=rank8u a=a a=bw b=b b=bw;
    ctrl=load;
sub8b:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=sub8b alu=rank8u a=a a=bw b=b b=bw;
    ctrl=load;
lsh8l:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=lsh8l alu=rank8u a=a a=bw;
    ctrl=load;
lsh8r:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=lsh8r alu=rank8u a=a a=bw;
    ctrl=load;
ash8l:
```

```
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=ash8l alu=rank8u a=a a=bw;
    ctrl=load;
ash8r:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=ash8r alu=rank8u a=a a=bw;
    ctrl=load;
rot8cl:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=rot8cl alu=rank8u a=a a=bw;
    ctrl=load;
rot8cr:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=rot8cr alu=rank8u a=a a=bw;
    ctrl=load;
rot8l:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=rot8l alu=rank8u a=a a=bw;
    ctrl=load;
rot8r:
    // C <= A[7:0]+B[7:0] (FL updated)
    fetch=f1 c=b c=load fl=b fl=load alu=rot8r alu=rank8u a=a a=bw;
    ctrl=load;
cast_uns_a:
    c=b c=load fl=b fl=load alu=a alu=rank8u a=a a=bw;
    fetch=f1 a=a a=load c=a c=bw
    ctrl=load;
cast_uns_b:
    c=b c=load fl=b fl=load alu=a alu=rank8u b=a b=bw;
    fetch=f1 b=a b=load c=a c=bw
    ctrl=load;
cast_sig_a:
    c=b c=load fl=b fl=load alu=a alu=rank8s a=a a=bw;
    fetch=f1 a=a a=load c=a c=bw
    ctrl=load;
cast_sig_b:
    c=b c=load fl=b fl=load alu=a alu=rank8s b=a b=bw;
    fetch=f1 b=a b=load c=a c=bw
    ctrl=load;
equals:
    // MAR <= SP, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // C <= MDR (FL updated)
```

```
    c=b c=load fl=b fl=load alu=a alu=rank16 mdr=a mdr=rank16 mdr=bw;
    fetch=f1; ctrl=load;
nots:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // C <= ~MDR (FL updated)
    c=b c=load fl=b fl=load alu=not alu=rank16 mdr=a mdr=rank16 mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
ands:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= A&B (FL updated)
    c=b c=load fl=b fl=load alu=and alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
nands:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= ~(A&B) (FL updated)
    c=b c=load fl=b fl=load alu=nand alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
```

```
    fetch=f1; ctrl=load;
ors:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= A|B (FL updated)
    c=b c=load fl=b fl=load alu=or alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
nors:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= ~(A|B) (FL updated)
    c=b c=load fl=b fl=load alu=nor alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
xors:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= A^B (FL updated)
```

```
    c=b c=load fl=b fl=load alu=xor alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
nxors:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= ~(A^B) (FL updated)
    c=b c=load fl=b fl=load alu=nxor alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
adds:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= A+B (FL updated)
    c=b c=load fl=b fl=load alu=add alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
subs:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
```

```
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= A-B (FL updated)
    c=b c=load fl=b fl=load alu=sub alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
addcs:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= A+B+carry (FL updated)
    c=b c=load fl=b fl=load alu=addc alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
subbs:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // C <= A-B-borrow (FL updated)
    c=b c=load fl=b fl=load alu=subb alu=rank16 a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
```

```
lshls:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // C <= lshl(A) (FL updated)
    c=b c=load fl=b fl=load alu=lshl alu=rank16 mdr=a mdr=rank16 mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
lshrs:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // C <= lshr(A) (FL updated)
    c=b c=load fl=b fl=load alu=lshr alu=rank16 mdr=a mdr=rank16 mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
ashls:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // C <= ashl(A) (FL updated)
    c=b c=load fl=b fl=load alu=ashl alu=rank16 mdr=a mdr=rank16 mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
ashrs:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // C <= ashr(A) (FL updated)
    c=b c=load fl=b fl=load alu=ashr alu=rank16 mdr=a mdr=rank16 mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
rotcls:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
```

```
    // C <= rotcl(A)  (FL updated)
    c=b c=load fl=b fl=load alu=rotcl alu=rank16 mdr=a mdr=rank16 mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
rotcrs:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // C <= rotcr(A)  (FL updated)
    c=b c=load fl=b fl=load alu=rotcr alu=rank16 mdr=a mdr=rank16 mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
rotls:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // C <= rotl(A)  (FL updated)
    c=b c=load fl=b fl=load alu=rotl alu=rank16 mdr=a mdr=rank16 mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
rotrs:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // C <= rotr(A)  (FL updated)
    c=b c=load fl=b fl=load alu=rotr alu=rank16 mdr=a mdr=rank16 mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
add8s:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= ++SP, MDR <= RAM (2 clocks);
```

```
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank8u mdr=bw;
    // C <= A+B (FL updated)
    c=b c=load fl=b fl=load alu=add8 alu=rank8u a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
sub8s:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank8u mdr=bw;
    // C <= A-B (FL updated)
    c=b c=load fl=b fl=load alu=sub8 alu=rank8u a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
add8cs:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank8u mdr=bw;
    // C <= A+B+carry (FL updated)
    c=b c=load fl=b fl=load alu=add8c alu=rank8u a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
sub8bs:
```

```
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank8u mdr=bw;
    // C <= A-B-borrow (FL updated)
    c=b c=load fl=b fl=load alu=sub8b alu=rank8u a=a a=bw b=b b=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
lsh8ls:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // C <= lsh8l(A) (FL updated)
    c=b c=load fl=b fl=load alu=lsh8l alu=rank8u mdr=a mdr=rank8u mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
lsh8rs:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // C <= lsh8r(A) (FL updated)
    c=b c=load fl=b fl=load alu=lsh8r alu=rank8u mdr=a mdr=rank8u mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
ash8ls:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // C <= ash8l(A) (FL updated)
    c=b c=load fl=b fl=load alu=ash8l alu=rank8u mdr=a mdr=rank8u mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
```

```
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
ash8rs:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // C <= ash8r(A) (FL updated)
    c=b c=load fl=b fl=load alu=ash8r alu=rank8u mdr=a mdr=rank8u mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
rot8cls:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // C <= rot8cl(A) (FL updated)
    c=b c=load fl=b fl=load alu=rot8cl alu=rank8u mdr=a mdr=rank8u mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
rot8crs:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // C <= rot8cr(A) (FL updated)
    c=b c=load fl=b fl=load alu=rot8cr alu=rank8u mdr=a mdr=rank8u mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
rot8ls:
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // C <= rot8l(A) (FL updated)
    c=b c=load fl=b fl=load alu=rot8l alu=rank8u mdr=a mdr=rank8u mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
rot8rs:
```

```
    // MAR <= ++SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=dp1; ctrl=nop;
    // C <= rot8r(A) (FL updated)
    c=b c=load fl=b fl=load alu=rot8r alu=rank8u mdr=a mdr=rank8u mdr=bw;
    // MDR <= C
    mdr=a mdr=load c=a c=bw;
    // MAR <= --SP, RAM <= MDR (2 clocks)
    mar=s mar=rank8 mar=load mar=ram_write sp=qm1 sp=qup; ctrl=nop;
    fetch=f1; ctrl=load;
casts_uns:
    // MAR <= SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // C <= MDR (FL updated)
    c=b c=load fl=b fl=load alu=a alu=rank8u mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
casts_sig:
    // MAR <= SP, MDR <= RAM (2 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // C <= MDR (FL updated)
    c=b c=load fl=b fl=load alu=a alu=rank8s mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
call:
    // MAR <= PC+=2, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // TMP <= MDR
    tmp=a tmp=load mdr=a mdr=rank16 mdr=bw;
    // MDR <= PC
    mdr=a mdr=load pc=a pc=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    // PC <= TMP
    pc=a pc=load tmp=a tmp=bw;
    fetch=f1; ctrl=load;
call_i:
    // MDR <= PC
    mdr=a mdr=load pc=a pc=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    // PC <= I
```

```
    pc=a pc=load i=a i=bw;
    fetch=f1; ctrl=load;
call_j:
    // MDR <= PC
    mdr=a mdr=load pc=a pc=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop; ctrl=nop;
    // PC <= J
    pc=a pc=load j=a j=bw;
    fetch=f1; ctrl=load;
return:
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // SP <= MDR;
    fetch=f1 sp=a sp=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
ivtl:
    // MAR <= PC+=2, MDR <= RAM (3 clocks);
    mar=p mar=rank16 mar=load pc=dp2 mdr=ram_read; ctrl=nop; ctrl=nop;
    // IVT <= MDR
    fetch=f1 ivt=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
int:
    //
    // push FL
    //
    // MDR <= FL
    mdr=a mdr=load fl=a fl=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop;
    //
    // Reset interrupt enable flag (while the MEM module is working).
    //
    fl=b fl=load alu=cleari;
    //
    // push (PC+1) to jump over the interrupt argument.
    //
    // MDR <= PC+1
    mdr=a mdr=load pc=a pc=qp1 pc=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop;
    //
    // Convert the interrupt number from the argument, to the
```

```
    // interrupt vector table item address:
    //     TMP <= IVT[int], int <= RAM[PC]
    //
    // MAR <= PC++, MDR <= RAM (2 clocks);
    mar=p mar=rank16 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // TMP <= IVT[int]
    tmp=a tmp=load ivt=int_b ivt=bw mdr=b mdr=rank8u mdr=bw;
    //
    // Prepare to call the interrupt routine, loading the
    // interrupt vector table item address into PC.
    //
    // MAR <= TMP, MDR <= RAM (3 clocks);
    mar=t mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // PC <= MDR;
    fetch=f1 pc=a pc=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
iret:
    //
    // Pop PC
    //
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // PC <= MDR;
    pc=a pc=load mdr=a mdr=rank16 mdr=bw;
    //
    // Pop FL
    //
    // MAR <= SP=+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=dp2; ctrl=nop; ctrl=nop;
    // FL <= MDR;
    fetch=f1 fl=a fl=load mdr=a mdr=rank16 mdr=bw;
    ctrl=load;
imrl:
    // MAR <= PC++, MDR <= RAM (2 clocks);
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // IRQ <= MDR;
    fetch=f1 irq=mask_load mdr=a mdr=rank8u mdr=bw;
    ctrl=load;
irq:
    //
    // push FL
    //
    // MDR <= FL
```

```
    mdr=a mdr=load fl=a fl=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop;
    //
    // Reset interrupt enable flag (while the MEM module is working).
    //
    fl=b fl=load alu=cleari;
    //
    // push (PC-1) to jump back to the opcode just read.
    //
    // MDR <= PC-1
    mdr=a mdr=load pc=a pc=qm1 pc=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop;
    //
    // Convert the interrupt number from the IRQ module, to the
    // interrupt vector table item address:
    //     TMP <= IVT[int], int <= IRQ
    //
    // TMP <= IVT[INT[IRQ]]
    tmp=a tmp=load ivt=int_a ivt=bw;
    //
    // Prepare to call the interrupt routine, loading the
    // interrupt vector table item address into PC.
    //
    // MAR <= TMP, MDR <= RAM (3 clocks);
    mar=t mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // PC <= MDR;
    pc=a pc=load mdr=a mdr=rank16 mdr=bw;
    //
    // Set the IRQ received as already done.
    //
    //
    fetch=f1 irq=irq_done;
    ctrl=load;
op_error:
    //
    // push FL
    //
    // MDR <= FL
    mdr=a mdr=load fl=a fl=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop;
```

```
    //
    // Reset interrupt enable flag (while the MEM module is working).
    //
    fl=b fl=load alu=cleari;
    //
    // push (PC-1) to jump back to the opcode just read.
    //
    // MDR <= PC-1
    mdr=a mdr=load pc=a pc=qm1 pc=bw;
    // MAR <= SP=-2, RAM <= MDR (3 clocks)
    mar=s mar=rank16 mar=load mar=ram_write sp=qm2 sp=qup; ctrl=nop;
    //
    // Convert the interrupt number 0 to the
    // interrupt vector table item address:
    //
    // TMP <= IVT[0]
    tmp=a tmp=load ivt=int_b ivt=bw bus=b bus=bw bus=0;
    //
    // Prepare to call the interrupt routine, loading the
    // interrupt vector table item address into PC.
    //
    // MAR <= TMP, MDR <= RAM (3 clocks);
    mar=t mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // PC <= MDR;
    pc=a pc=load mdr=a mdr=rank16 mdr=bw;
    fetch=f1;
    ctrl=load;
flag_i_clr:
    // FL <= FL&0xFFEF
    fetch=f1 fl=b fl=load alu=cleari;
    ctrl=load;
flag_i_set:
    // FL <= FL|0x0010
    fetch=f1 fl=b fl=load alu=seti;
    ctrl=load;
flag_c_clr:
    // FL <= FL&0xFFFE
    fetch=f1 fl=b fl=load alu=clearc;
    ctrl=load;
flag_c_set:
    // FL <= FL|0x0001
    fetch=f1 fl=b fl=load alu=setc;
    ctrl=load;
```

```
in_num_a:
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // I/O request (2 clocks)
    ioc=req; ctrl=nop;

    ctrl=nop;
    ctrl=nop;

    // A <= IOC

    a=a a=load ioc=bw;

    ctrl=nop;
    ctrl=nop;

    fetch=f1;

    ctrl=load;
in_num_b:
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp2 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // I/O request (2 clocks)
    ioc=req; ctrl=nop;
    // B <= IOC
    fetch=f1 b=a b=load ioc=bw;
    ctrl=load;
in_num_mi:
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp2 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // I/O request (2 clocks)
    ioc=req; ctrl=nop;
    // MDR <= IOC
    mdr=a mdr=load ioc=bw;
    // MAR <= I, RAM <= MDR (3 clocks)
    mar=i mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
```

```
in_num_mj:
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp2 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // I/O request (2 clocks)
    ioc=req; ctrl=nop;
    // MDR <= IOC
    mdr=a mdr=load ioc=bw;
    // MAR <= J, RAM <= MDR (3 clocks)
    mar=j mar=rank16 mar=load mar=ram_write; ctrl=nop; ctrl=nop;
    fetch=f1; ctrl=load;
out_a_num:
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // IOC <= A
    ioc=load a=a a=bw;
    // I/O request
    ioc=req
    fetch=f1; ctrl=load;
out_b_num:
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // IOC <= B
    ioc=load b=a b=bw;
    // I/O request
    ioc=req
    fetch=f1; ctrl=load;
out_mi_num:
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // MAR <= I, MDR <= RAM (3 clocks);
    mar=i mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // IOC <= MDR;
    ioc=req
    fetch=f1; ctrl=load;
out_mj_num:
```

```
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // MAR <= J, MDR <= RAM (3 clocks);
    mar=j mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // IOC <= MDR;
    ioc=req
    fetch=f1; ctrl=load;
ifack_jump:
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // C <= alu(I/O isack) (FL updated) and conditional jump.
    c=b c=load fl=b fl=load alu=a alu=rank8s ioc=isack ioc=bw
    addr=jump jump=zero_f;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
ifack_call:
    // MAR <= PC++, MDR <= RAM (2 clocks)
    mar=p mar=rank8 mar=load pc=dp1 mdr=ram_read; ctrl=nop;
    // IOA <= MDR
    ioa=a ioa=load mdr=a mdr=rank8u mdr=bw;
    // C <= alu(I/O isack) (FL updated) and conditional jump.
    c=b c=load fl=b fl=load alu=a alu=rank8s ioc=isack ioc=bw
    addr=call jump=zero_f;
    // discard destination address: ++PC and fetch
    pc=qp2 pc=qup fetch=f1;
    ctrl=load;
cmpr:
    // FL(A-B)
    fetch=f1 fl=b fl=load alu=sub alu=rank16;
    ctrl=load;
testr:
    // FL(A&B), MPC <= #fetch
    fetch=f1 fl=b fl=load alu=and alu=rank16;
    ctrl=load;
cmpi:
    // MAR <= I, MDR <= RAM (3 clocks);
    mar=i mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // A <= MDR;
```

```
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= J, MDR <= RAM (3 clocks);
    mar=j mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw
    addr=cmpr jump=true;
testi:
    // MAR <= I, MDR <= RAM (3 clocks);
    mar=i mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= J, MDR <= RAM (3 clocks);
    mar=j mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw
    addr=testr jump=true;
cmps:
    // MAR <= SP, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=qp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw
    addr=cmpr jump=true;
tests:
    // MAR <= SP, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank16 mdr=bw;
    // MAR <= SP+2, MDR <= RAM (3 clocks);
    mar=s mar=rank16 mar=load mdr=ram_read sp=qp2; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank16 mdr=bw
    addr=testr jump=true;
cmp8i:
    // MAR <= I, MDR <= RAM (2 clocks);
    mar=i mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank8s mdr=bw;
    // MAR <= J, MDR <= RAM (2 clocks);
    mar=j mar=rank8 mar=load mdr=ram_read; ctrl=nop;
```

```
    // B <= MDR;
    b=a b=load mdr=a mdr=rank8s mdr=bw;
    addr=cmpr jump=true;
test8i:
    // MAR <= I, MDR <= RAM (2 clocks);
    mar=i mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank8s mdr=bw;
    // MAR <= J, MDR <= RAM (2 clocks);
    mar=j mar=rank8 mar=load mdr=ram_read; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank8s mdr=bw;
    addr=and jump=true;
cmp8s:
    // MAR <= SP, MDR <= RAM (3 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank8s mdr=bw;
    // MAR <= SP+1, MDR <= RAM (3 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=qp1; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank8s mdr=bw
    addr=cmpr jump=true;
test8s:
    // MAR <= SP, MDR <= RAM (3 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read; ctrl=nop; ctrl=nop;
    // B <= MDR;
    b=a b=load mdr=a mdr=rank8s mdr=bw;
    // MAR <= SP+1, MDR <= RAM (3 clocks);
    mar=s mar=rank8 mar=load mdr=ram_read sp=qp1; ctrl=nop; ctrl=nop;
    // A <= MDR;
    a=a a=load mdr=a mdr=rank8s mdr=bw
    addr=testr jump=true;
//
stop:
    ctrl=stop;
//
end
```

12.15 Macrocode

As an example, the following listing shows a simple program that reads from the keyboard and prints the same text on the virtual screen. The keyboard input is read after a hardware interrupt.

Listing 12.44. Macrocode example for TKGate.

```
begin macrocode @ 0
        jump #start
        nop
interrupt_vector_table:
        .short  0x0025  // CPU opcode error
        .short  0x0024  // CPU
        .short  0x0024  // CPU
        .short  0x0024  // CPU
        .short  0x0024  // IRQ rtc
        .short  0x0026  // IRQ keyboard
        .short  0x0024  // IRQ hard disk
        .short  0x0024  // IRQ
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
        .short  0x0024  // software
default_interrupt_routine:
        iret
op_code_error:
        stop
keyboard:
        in 1,%A                         // keyboard read
        equal                           // update flags
        jump %zero,1,#keyboard_end      // if zero exit
        out %A,0                        // print on screen
        jump #keyboard                  // continue
keyboard_end:
```

```
        iret
start:
        cp 0x0080, %I                       // set stack bottom
        cp %I, %SP
        //
        ivtl #interrupt_vector_table
        //
        imrl 0x0F                           // all IRQ accepted
        flag_i 1                            // IRQ enabled
//
keyboard_reset:
        in 1,%A                             // keyboard read.
        equal                               // update flags
        jump %zero,0,#keyboard_reset        // if not zero continue
//
loop:
        jump #loop
stop:                   // never reach the end.
        stop
end
```

13.1 General purpose modules 850
13.2 Addition modules 853
13.3 Multiple clock lines 857
13.4 Bus 860
13.5 Registers: simple and with post-increment 862
13.6 Memory segmentation 863
13.7 Module "DA" 864
13.8 Module "CA" 865
13.9 Modules "M" and "RAM" 866
13.10 Module "IR" 871
13.11 Module "MD" 873
13.12 Module "IRQ" 875
13.13 Module "IVT" 877
13.14 Module "ALU" 878
13.15 Module "FL" 887
13.16 Module "TTY" 890
13.17 Module "HDD" 894
13.18 Module "CTRL" 899
13.19 Memory and microcode fields 902
13.20 Opcodes 904
13.21 Data initialization 934
13.22 Microcode 935
13.23 Process mode and interrupts 966
13.24 Macrocode example: 'memcpy' 967

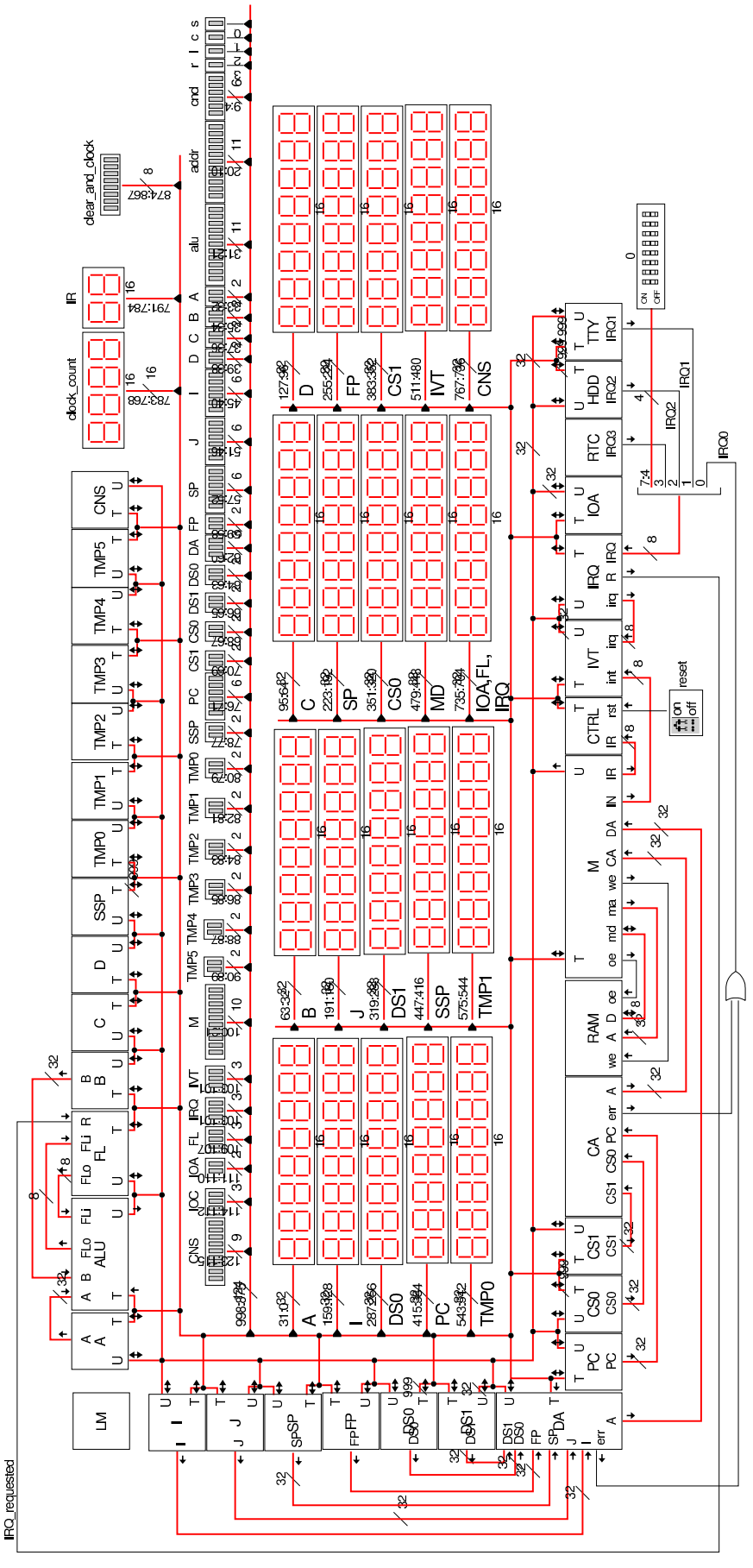
"from the ground up" draft-2013.03 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://a3.informaticalibera.net

13.25 Macrocode example: TTY with interrupt 969
13.26 Macrocode example: user and supervisor 972

This version of the CPU is made with 32-bit registers and is able to distinguish between privileged and unprivileged processes: unprivileged processes cannot access directly the I/O devices and cannot modify some status flags. The opcode remains limited to 8-bit width, including the argument represented by registers: because of this limitation, there are few macrocode instructions, but enough for the demonstration purpose of the project.

Attachments	Description
attachments/xcpu/xcpu-f.v	TKGate Verilog netlist source file.
attachments/xcpu/xcpu-f-memcpy.gm *attachments/xcpu/xcpu-f-tty-simple.gm* *attachments/xcpu/xcpu-f-user-process.gm* *attachments/xcpu/xcpu-f-supervisor-and-user.gm*	TKGate microcode and macrocode alternative source files.
attachments/xcpu/xcpu-f-terminal.vpd.tcl	TCL/Tk script related to the terminal module **TTY**.

Figure 13.2. Simple CPU, version “F”.



13.1 General purpose modules

The following figures show some general purpose modules. Some of them already appeared in previous versions of the project, with or without modifications.

Figure 13.3. Module **RANK32** and submodules: rank reduction and sign extention.

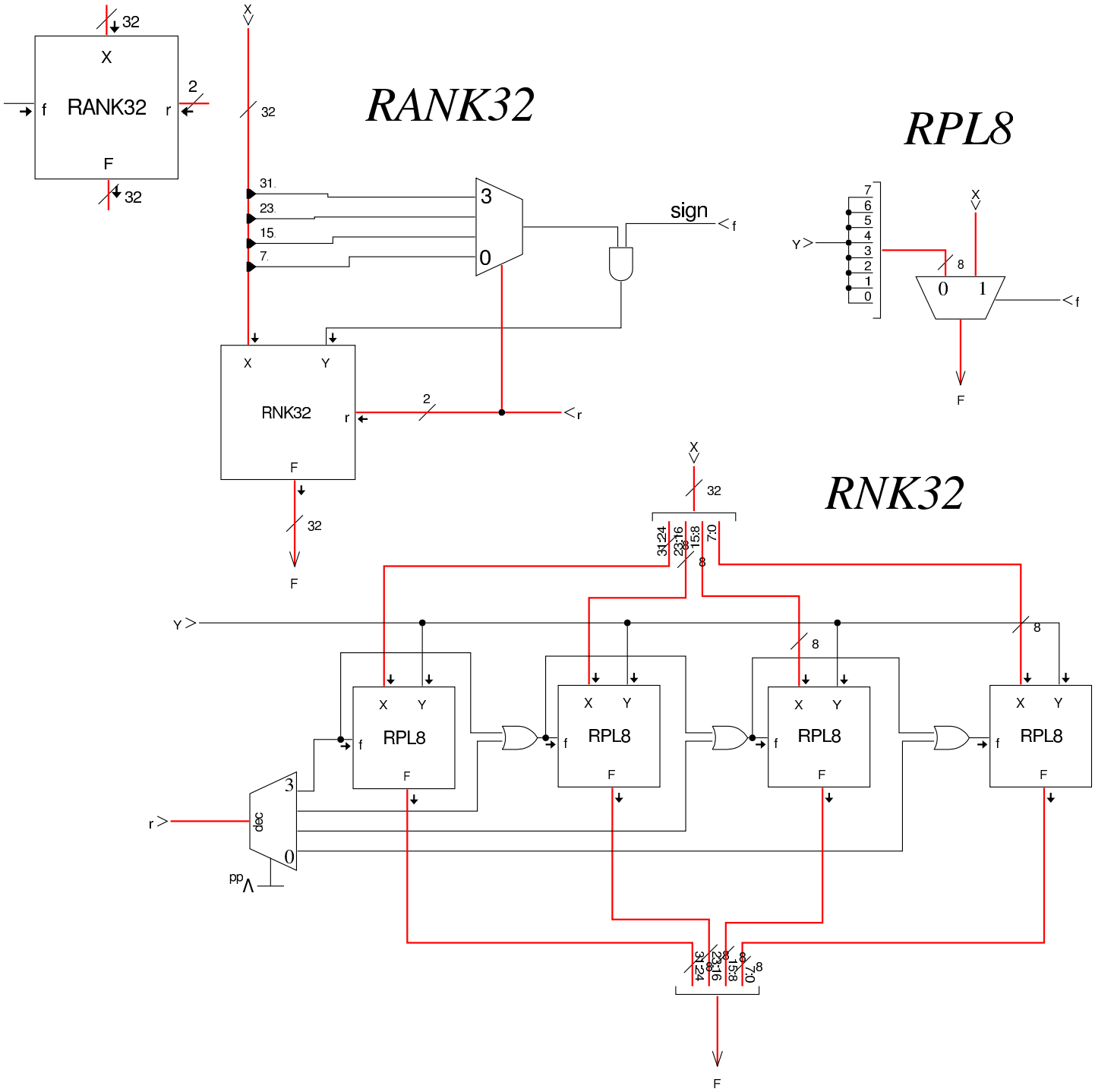


Figure 13.4. Modules **D*n***.

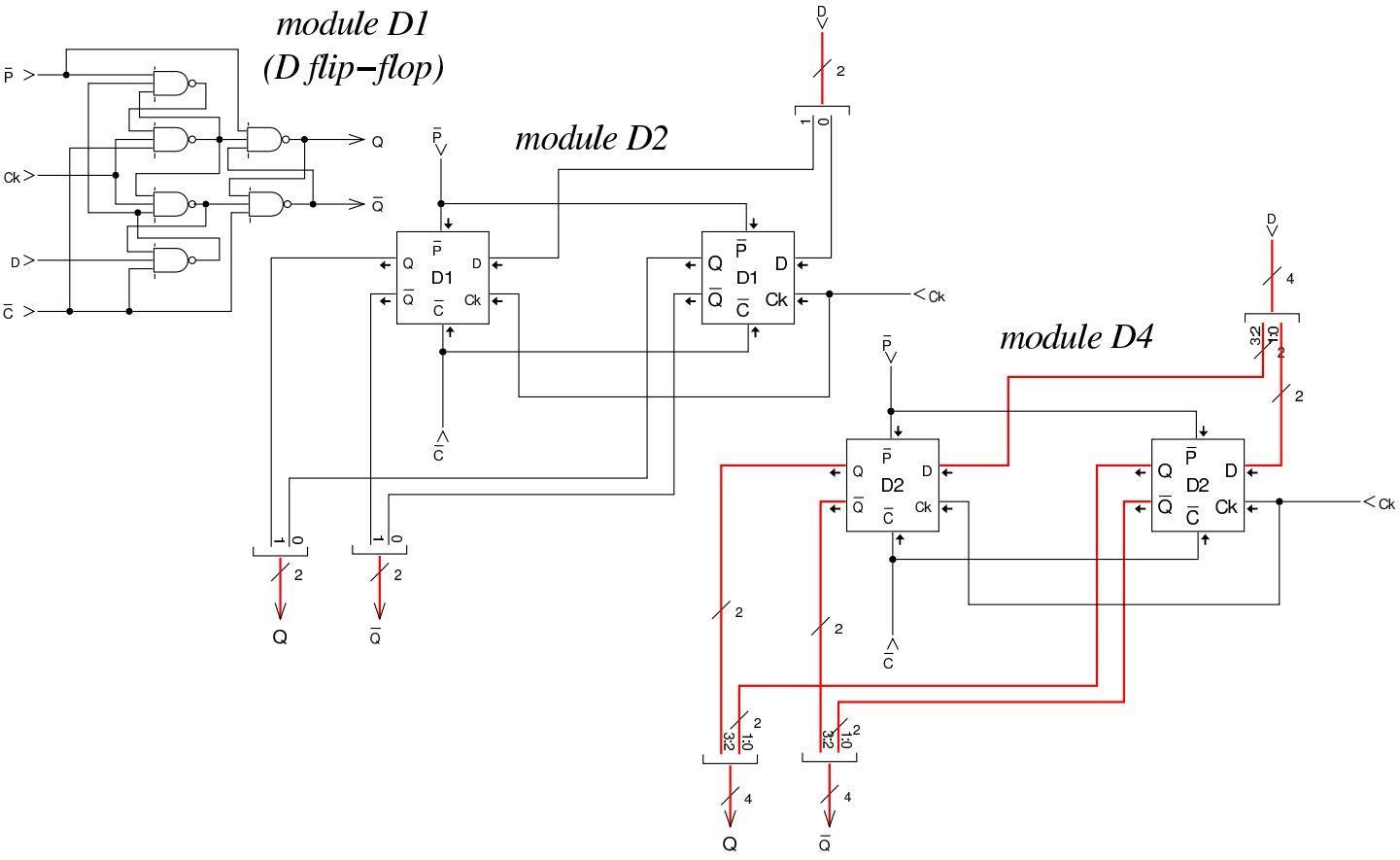


Figure 13.5. Modules **DH*n***.

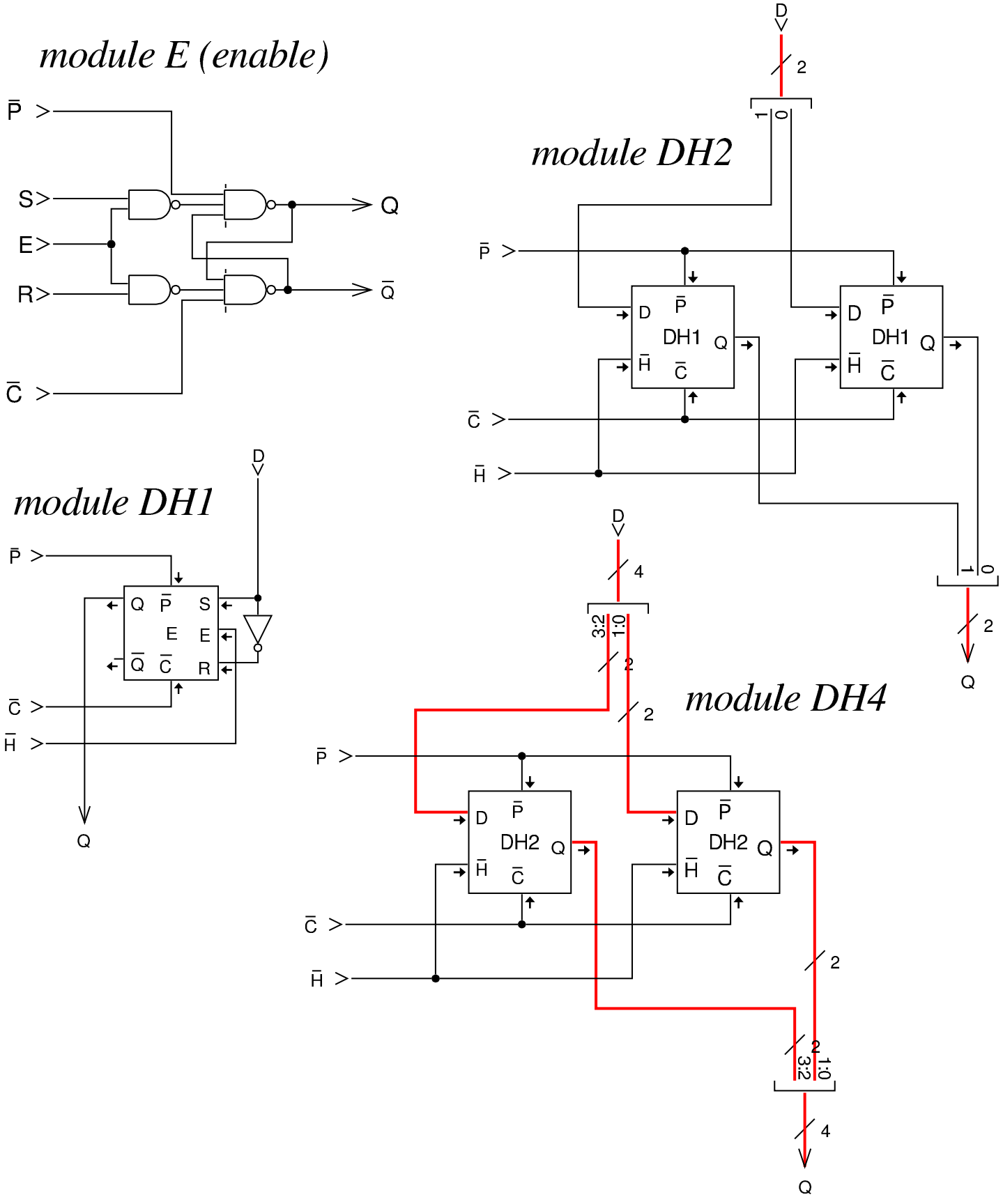
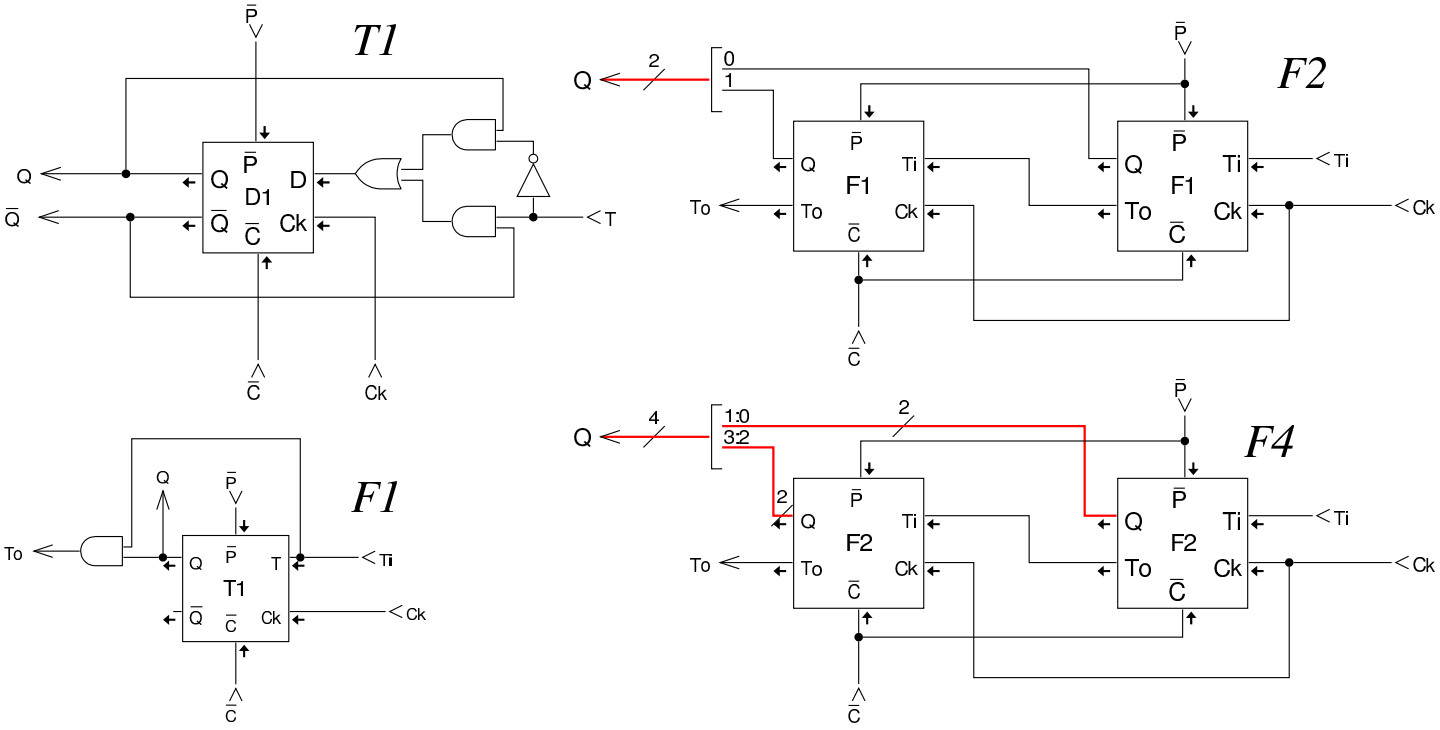


Figure 13.6. Modules **Fn**. A module **Fn**, provided that the input ***Ti*** is asserted, returns the count of the impulses received at the ***Ck*** input.



13.2 Addition modules

This version of the simple CPU works with 32-bit registers and ALU. The ALU, some registers and some other modules contain an addition module for various reasons. For example **DA** and **CA** need to calculate the real memory address and some registers need to be incremented or decremented. To ensure that the additions are made as fast as possible, the technique of the carry lookahead is used.

Figure 13.7. Module **SUM8**: 8-bit addition with carry lookahead.

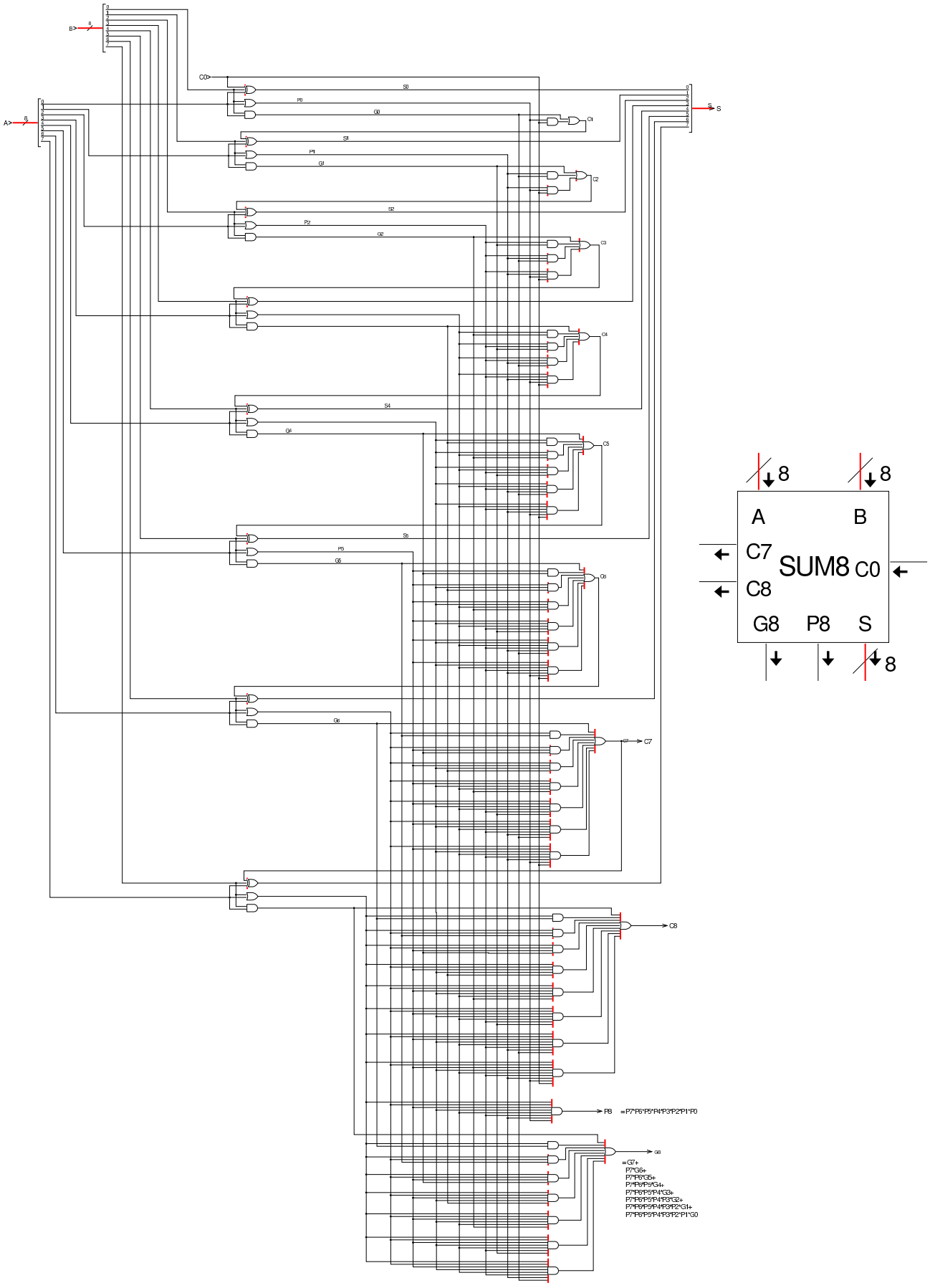


Figure 13.8. Module **`CLH4`** (*carry lookahead*).

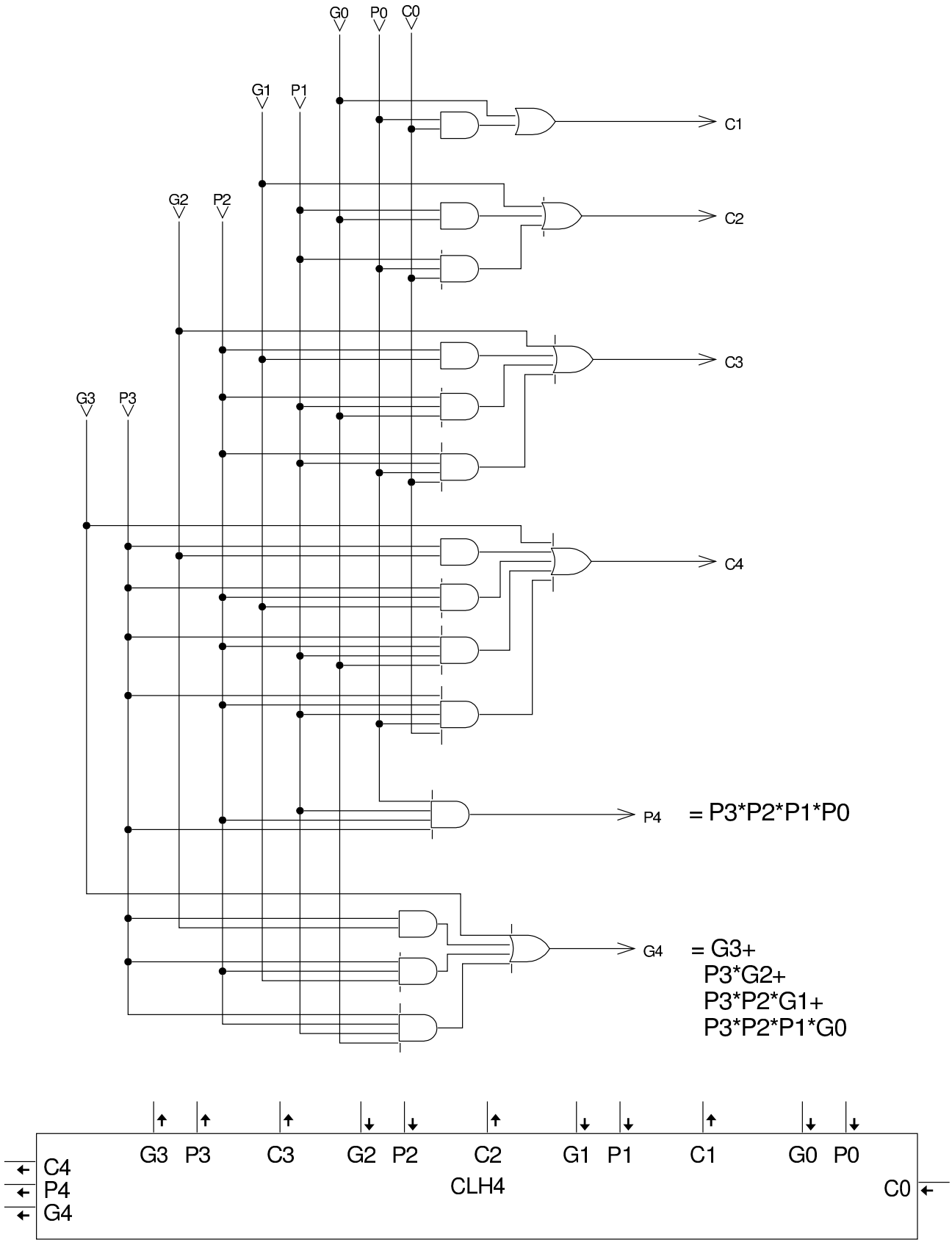


Figure 13.9. Module **SUM32**.

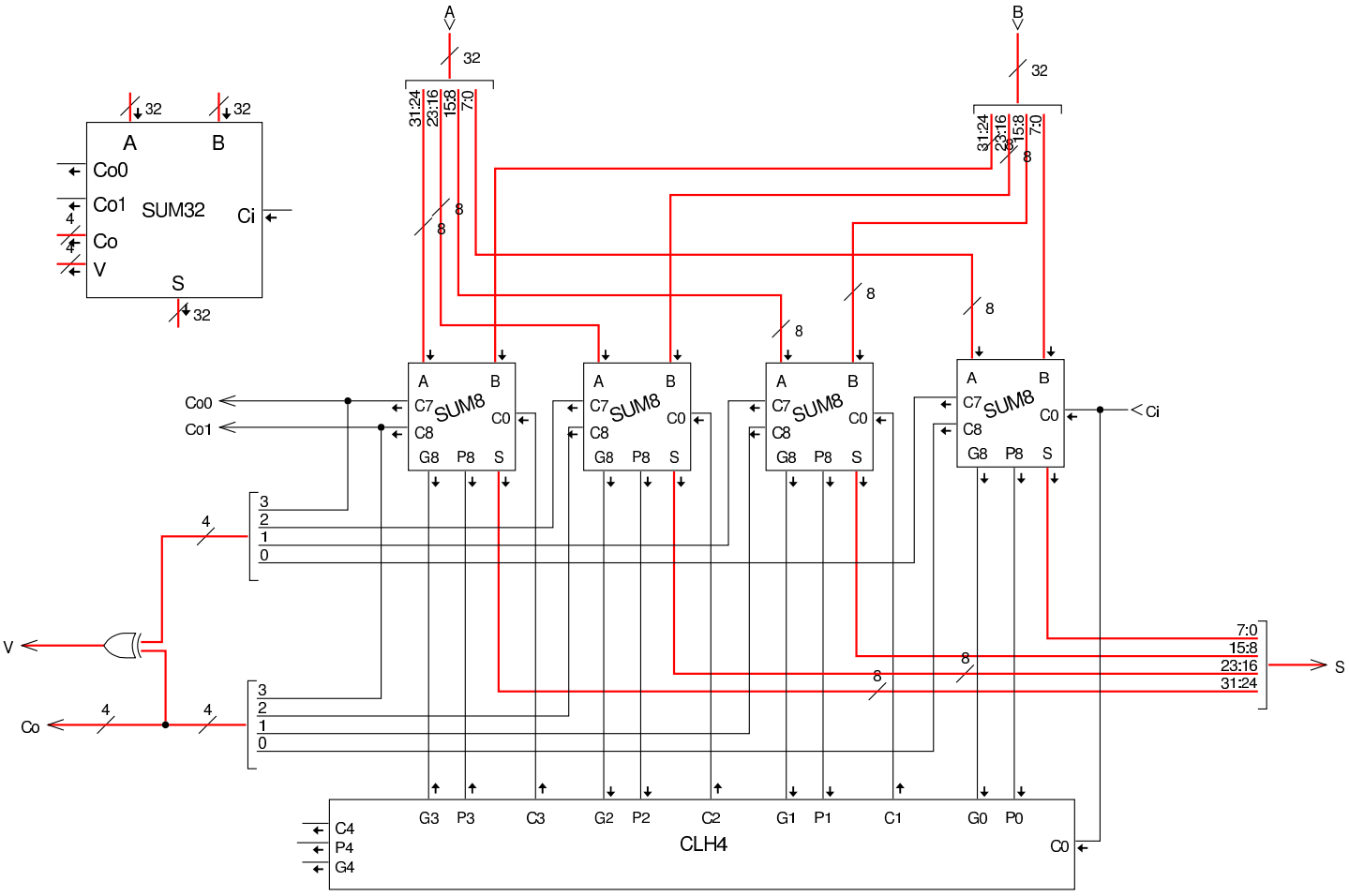
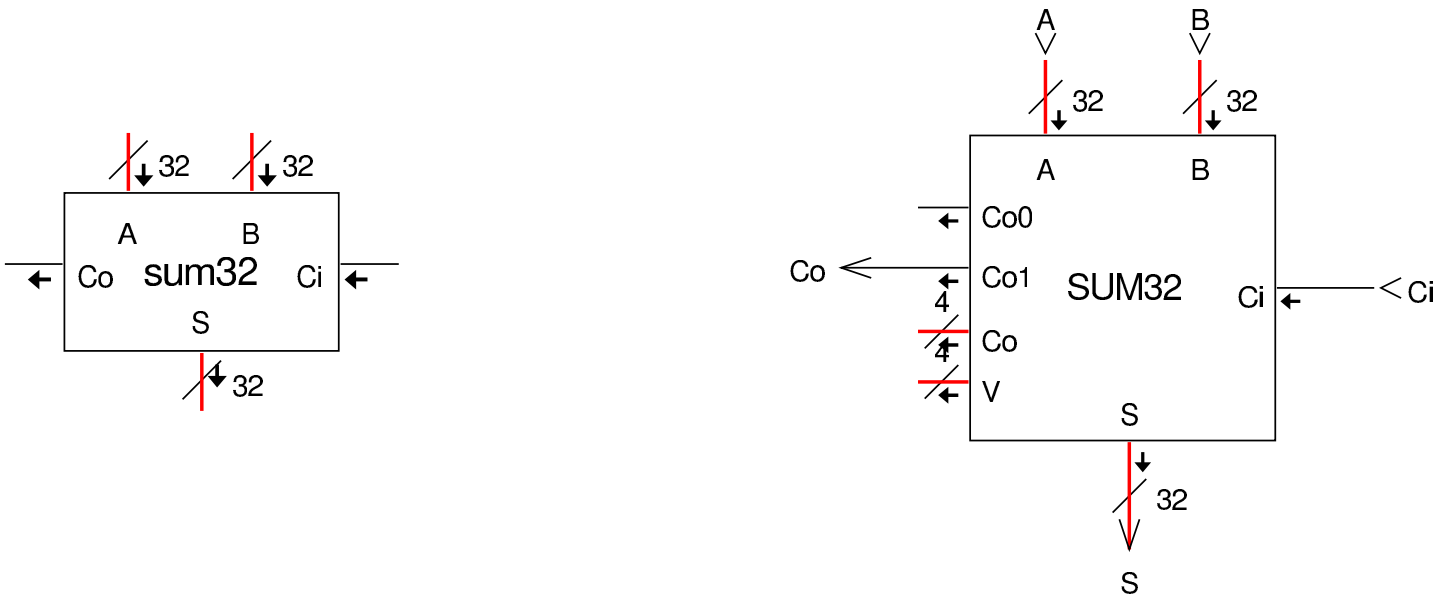


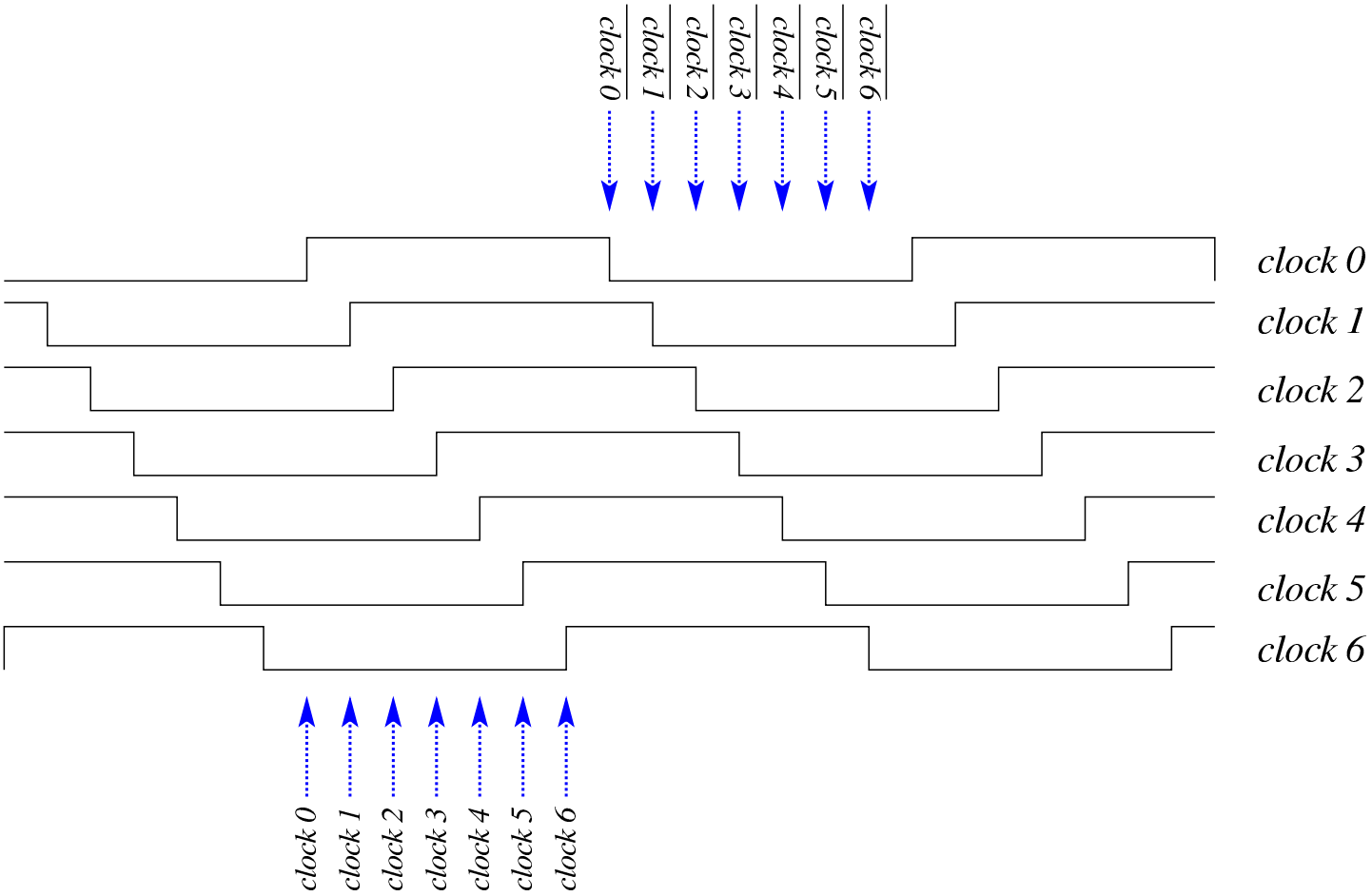
Figure 13.10. Module **sum32**: it is just a wrapper around the module **SUM32**.



13.3 Multiple clock lines

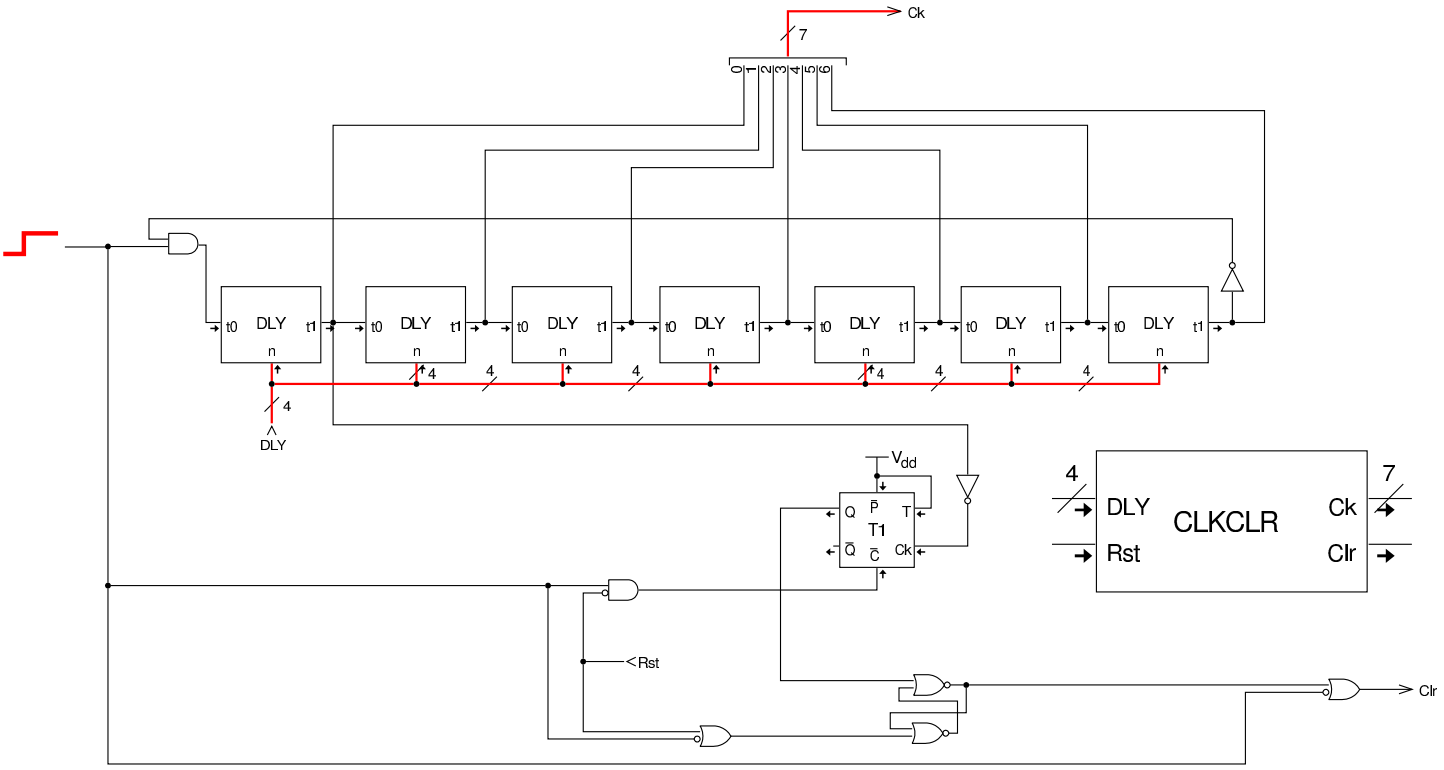
All the previous versions of the simple CPU suffer from clock timing problems that sometimes arise with the memory access. With this version there is a clock generating seven pulses at different offsets: with this method, up to fourteen edge triggers can be selected.

Figure 13.11. Seven clock signals.



The clock pulse is produced by the module **CLKCLR**, described by the following figures.

Figure 13.12. Module **CLKCLR**.

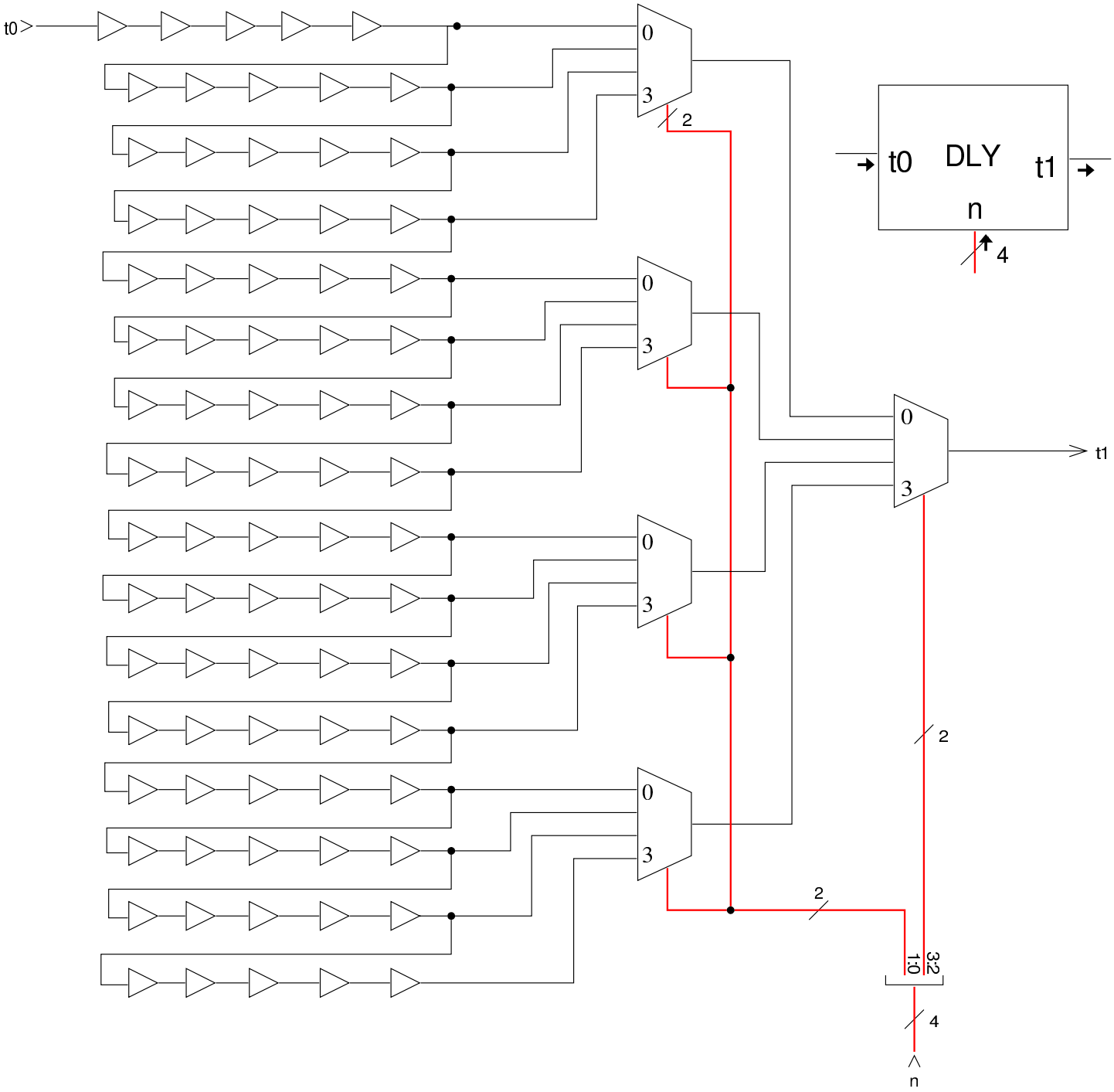


As it is for the previous version of the simple CPU, the clock generator is started by a single shot (the module **one_up**), represented on the left of the figure by a red 'stair'. Then the modules **DLY** are simple delay lines, made of non-inverting buffers. To tune the frequency, the delay lines can be adapted with the input value ***DLY***: zero represents the maximum frequency (the shortest delay), 15 produces the minimum frequency (the longest delay). As for the previous clock versions, the clear line (***Clr*** output) is to be coordinated with the clock.

Listing 13.13. Module **one_up**, used to start the oscillation inside the module **CLKCLR**.

```
module one_up #(.W(1000)) (Z);
  output Z;
  reg Z;

  initial
    begin
      Z = 1'b0;
      $tkg$wait(W);
      Z = 1'b1;
    end

endmodule
```

Figure 13.14. Module **DLY**.



13.4 Bus

This version of the simple CPU has only two busses: the ***U*** bus is the usual data bus with 32 bits, whereas the ***T*** is a double purpose bus, used to carry the content of all registers and the microcode control word. In practice: the ***U*** bus is used to exchange data, while the ***T*** bus allows to any component to read any other register,

without extra cabling. In fact, the project could have been designed with much less external cabling, but extra visible connections are used also to show some important dependencies between modules.

Inside the ***T*** bus, $\boldsymbol{T}_{791:0}$ is used to carry a copy of the content from every register, as it is described by the following table.

Table 13.15. $\boldsymbol{T}_{791:0}$ fields.

msb	lsb	size	value of:	msb	lsb	size	value of:
31	0	32	A	479	448	32	MD
63	32	32	B	511	480	32	IVT
95	64	32	C	543	512	32	TMP0
127	96	32	D	575	544	32	TMP1
159	128	32	I	607	576	32	TMP2
191	160	32	J	639	608	32	TMP3
223	192	32	SP	671	640	32	TMP4
255	224	32	FP	703	672	32	TMP5
287	256	32	DS0	719	704	16	IRQ
319	288	32	DS1	727	720	8	FL
351	320	32	CS0	735	728	8	IOA
383	352	32	CS1	767	736	32	CNS
415	384	32	PC	783	768	16	CLKC
447	416	32	SSP	791	784	8	IR

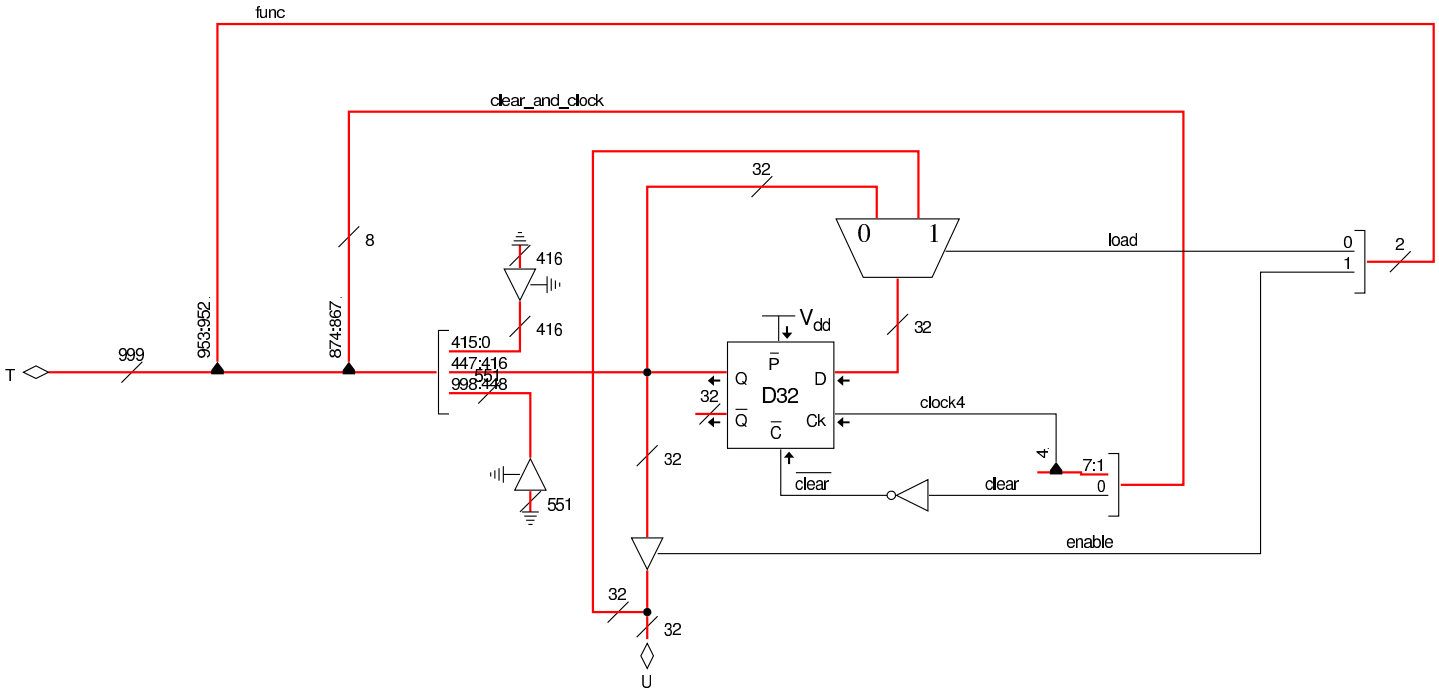
For example, the content of the ***FP*** register is available from $\boldsymbol{T}_{255:224}$. The following table shows the lines from bit 867 to bit 999, used to carry the functions to be executed by the modules connected to the ***T*** bus: on the left side there are the connections to the ***T*** bus, whereas on the right side there are the original partitions from the microcode word. Please notice that, for this version of the simple CPU, there are eight lines for clear and clock lines, which are added to the microcode word itself without replacing any line from the original microcode word.

Table 13.16. $T_{998:867}$ fields and $\mu CW_{123:0}$.

msb	lsb	size	line description	msb	lsb	msb	lsb	size	line description	msb	lsb
874	867	8	clear and clock								
878	875	4	stop, call, load, return	3	0	945	944	2	CS1	70	69
884	879	6	µC jump condition	9	4	951	946	6	PC	76	71
895	885	11	µC jump address	20	10	953	952	2	SSP	78	77
906	896	11	ALU	31	21	955	954	2	TMP0	80	79
908	907	2	A	33	32	957	956	2	TMP1	82	81
910	909	2	B	35	34	959	958	2	TMP2	84	83
912	911	2	C	37	36	961	960	2	TMP3	86	85
914	913	2	D	39	38	963	962	2	TMP4	88	87
920	915	6	I	45	40	965	964	2	TMP5	90	89
926	921	6	J	51	46	975	966	10	M	100	91
932	927	6	SP	57	52	978	976	3	IVT	103	101
934	933	2	FP	59	58	981	979	3	IRQ	106	104
937	935	3	DA	62	60	984	982	3	FL	109	107
939	938	2	DS0	64	63	986	985	2	IOA	111	110
941	940	2	DS1	66	65	989	987	3	IOC	114	112
943	942	2	CS0	68	67	998	990	9	CNS	123	115

13.5 Registers: simple and with post-increment

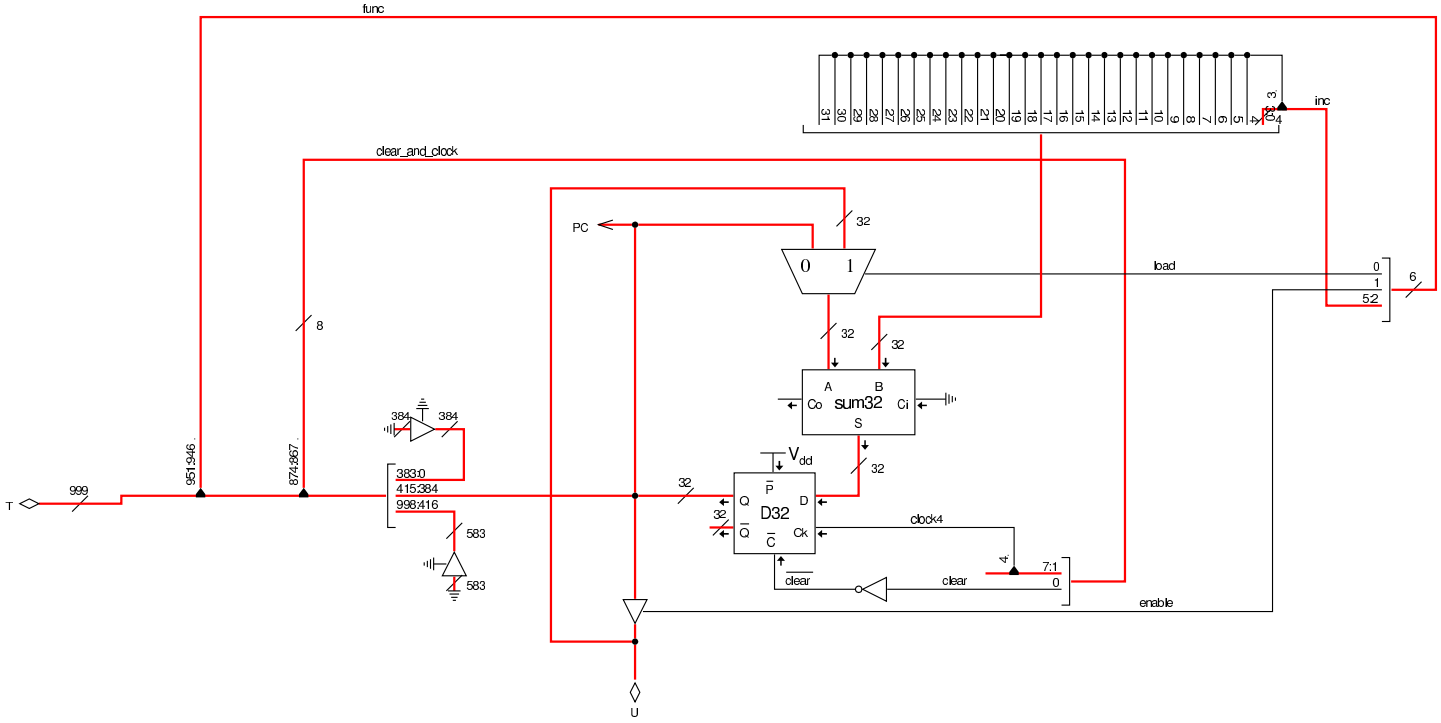
There are two types of registers, the registers ***A***, ***B***, ***C***, ***D***, ***FP***, ***CS0***, ***CS1***, ***DS0***, ***DS1***, ***SSP***, ***TMP0***, ***TMP1***, ***TMP2***, ***TMP3***, ***TMP4***, ***TMP5*** and ***IOA*** are simple registers like the one shown in the following figure, which corresponds to ***SSP***:



The register is made with a 32-bit D flip-flop (positive edge triggered), connected to a multiplexer that can select from the previous flip-flop output or a new input from the ***U*** bus. The control lines are only two: ***load*** (bus-read) and ***enable*** (bus-write). The

control lines (the function) are taken from the ***T*** bus (in that case they correspond to $\boldsymbol{T}_{953:952}$) and a copy of the register content is also sent to the ***T*** bus ($\boldsymbol{T}_{447:416}$).

The registers ***I***, ***J***, ***SP*** and ***PC***, can be post-incremented or post-decremented. The following figure shows the register ***PC***:



In that case the control lines are six: ***load*** (bus-read), ***enable*** (bus-write) and four lines representing a signed increment, from −8 to +7.

Some registers (***A***, ***B***, ***I***, ***J***, ***SP***, ***FP***, ***DS0***, ***DS1***, ***CS0***, ***CS1*** and ***PC***) have an extra output connection (like it appears in the previous figure). This is done to connect the register to some other module. As already explained in a previous section, every register sends its own value to a private channel inside the ***T*** where it could be read by any other module connected to it. But here these extra connections are kept to highlight their purpose.

13.6 Memory segmentation

In this version of the simple CPU, the address used to access the memory is relative to a 'segment'. That is: if a program is working inside a segment located at address 100000_{16}, up to address 200000_{16}, the relative address 123_{16} is located in memory at the real address 100123_{16}, moreover, if the programs tries to reach the location 100001_{16}, which corresponds to the real address 200001_{16}, an interrupt if fired and the program might be stopped.

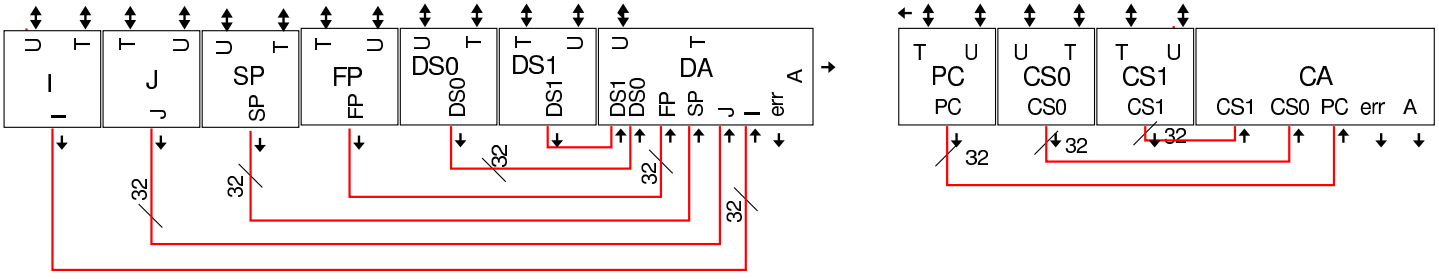
There are two memory segments: data and code. The data segment is delimited by the registers ***DS0*** and ***DS1***, where the former is used to point to the start of the segment, whereas the latter should point to the address after the end of the segment. For example, if ***DS0*** and ***DS1*** contains zero (which is the default starting value), the segment covers the entire memory addressable. The code segment, represented by the registers ***CS0*** and ***CS1***, works the same way.

When accessing the memory, through the module **M**, it must be specified if the access is relative to the data segment, or the code segment. When loading the opcode inside the instruction register (***IR***) and the arguments following the opcode, the code segment is taken into consideration, whereas, for any other access, the data segment is in control.

The module **DA** (*data address*) is responsible for calculating the real address used for the data segment. It is connected to the registers ***I***, ***J***, ***SP*** and ***FP***: any data segment access requires to have loaded, at least, an address inside one of these registers.

The module **CA** (*code address*) is used to calculate the real address for the code segment and depends from the register ***PC***.

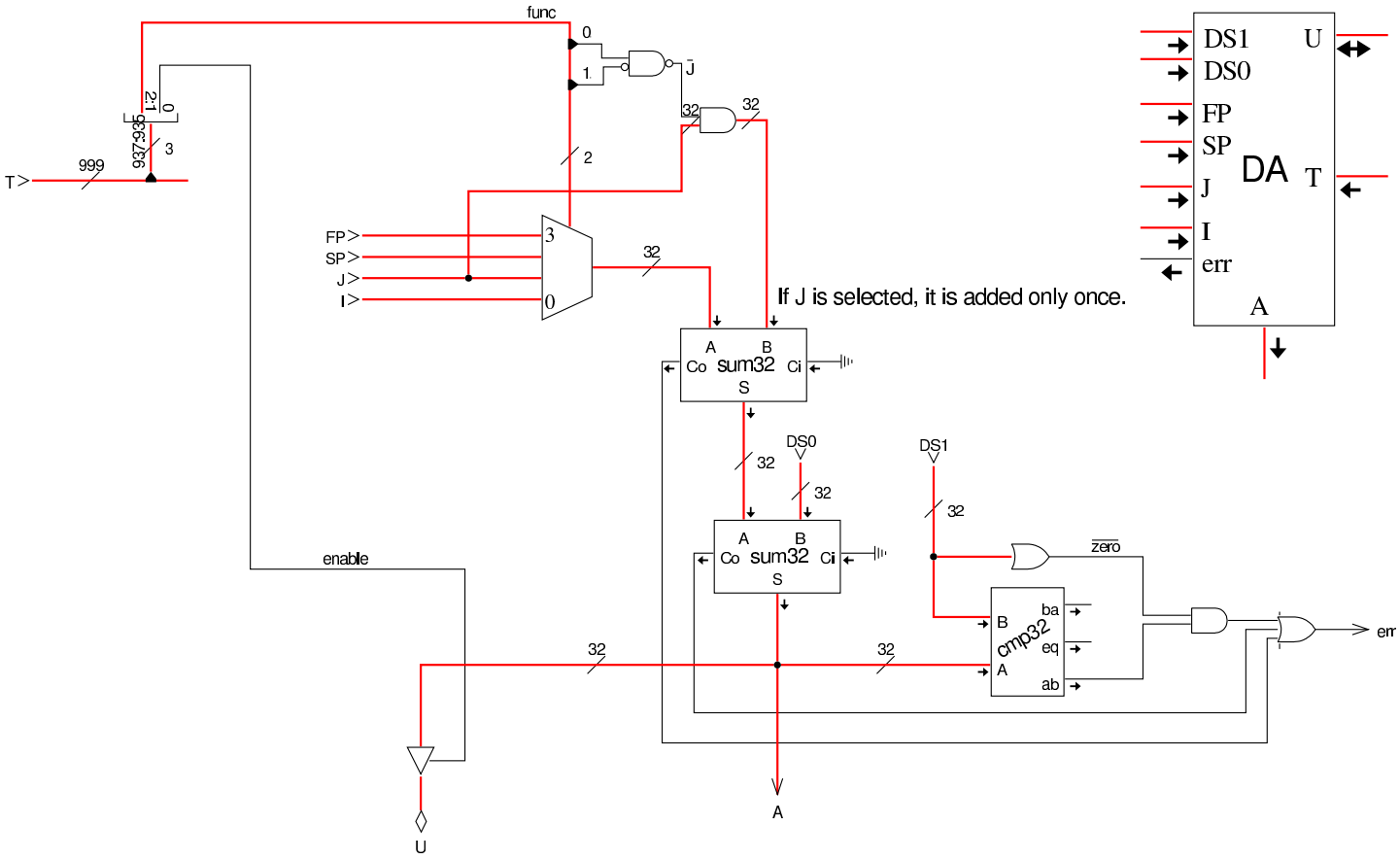
Figure 13.19. Connection with modules **DA** and **CA**, to calculate and validate the requested addresses, for data and code segments.



13.7 Module "DA"

The module **DA** is used to calculate the real memory address for something located in the data segment. The real memory address is calculated adding the value contained inside the registers ***DS0*** and ***J***: the former is used to specify the begin of the data segment and the latter as an index from the selected relative address. The relative address used as input to the **DA** module might come from the address registers, ***I***, ***J***, ***SP*** and ***FP***, with the exception that if the ***J*** input is selected, ***J*** itself is not added again.

Figure 13.20. Module **DA**.

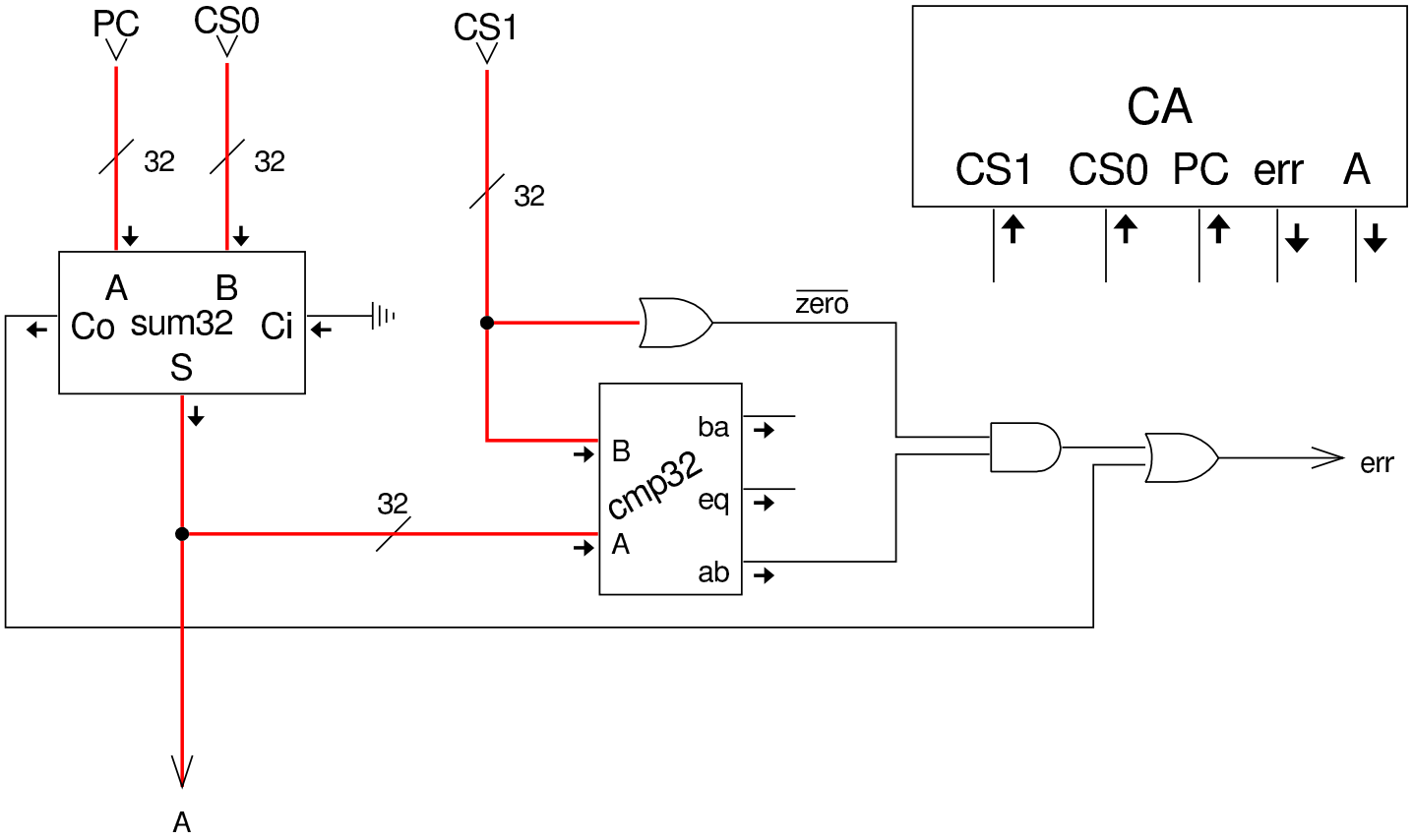


The module output is ***A*** (*address*) and a check is made to verify if the real memory address is contained before the value reported by register ***DS1***. If the calculated real address is outside the allowed memory segment, the output ***err*** is asserted. This version of the **DA** module does not prevent the access outside the allowed memory segment.

13.8 Module "CA"

The module **CA** is used to calculate the real memory address for the code. The real memory address is calculated adding the value contained inside the registers ***CS0*** and ***PC***.

Figure 13.21. Module **CA**.

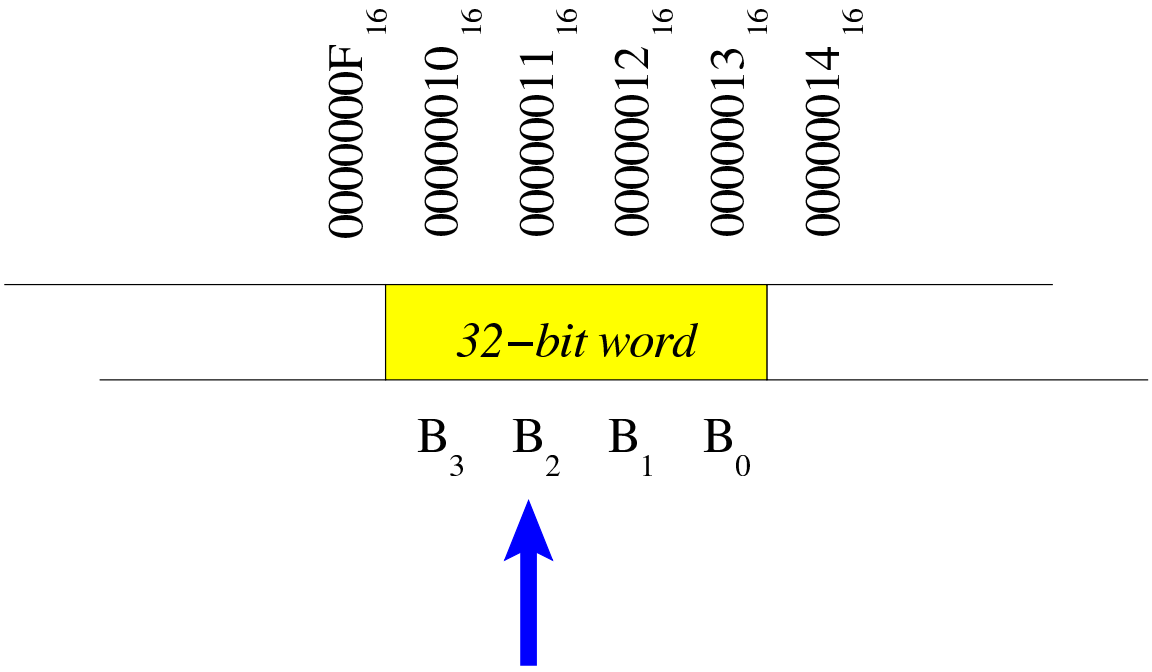


The module output is ***A*** (*address*) and a check is made to verify if the real memory address is contained before the value reported by register ***CS1***. If the calculated real address is outside the allowed memory segment, the output ***err*** is asserted. This version of the **CA** module does not prevent the access outside the allowed memory segment.

13.9 Modules "M" and "RAM"

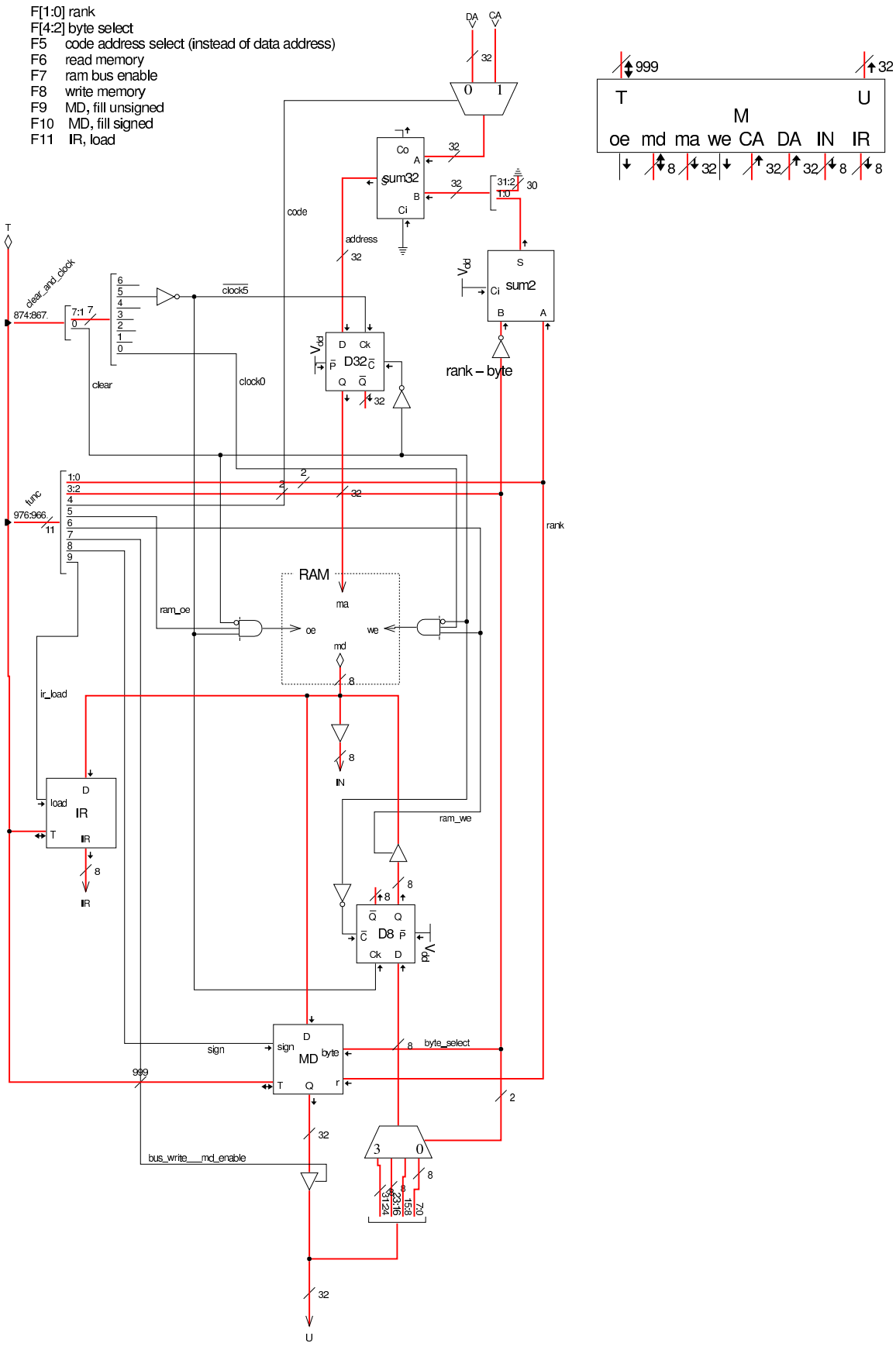
«

The module **M** is responsible for all memory access, selecting a real address from the **DA** mode or from the **CA** module. The memory is organized with 8-bit blocks, but the **M** module receives the request to access a certain byte inside a particular word that starts at a particular address. For example, it might be requested to access the byte two of a 32-bit word, located at the address 00000010_{16}:

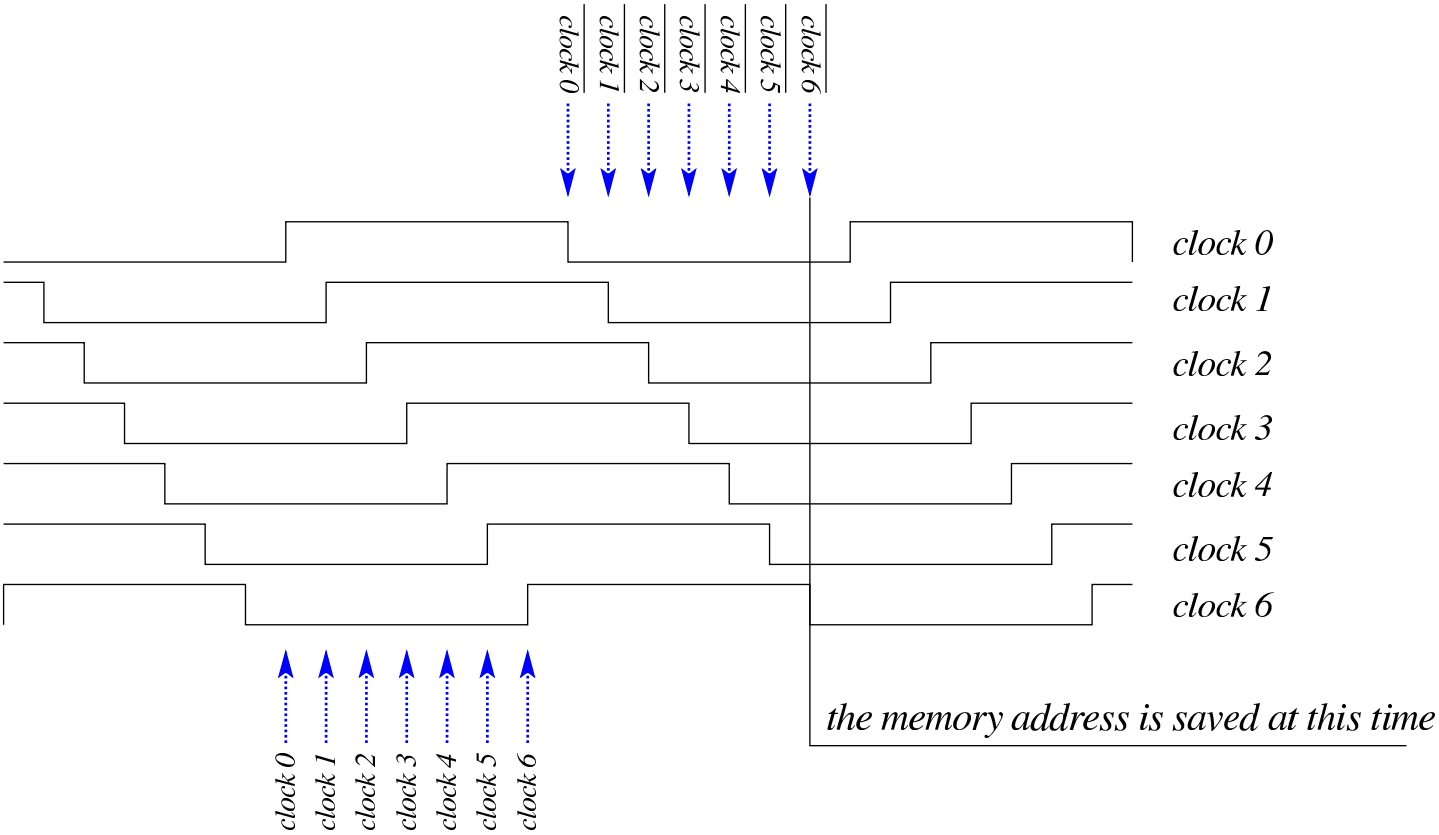


As it should be visible in the figure, the byte that the module **M** will access is located at address 00000011_{16}. To obtain this result, the module requires to know the width of the data requested (the rank) and the byte position inside the requested word. The rank is a number —from zero to three— that refers to a rank from one byte to four bytes; the byte position is a number —from zero to three— that refers to a byte from the least significant byte (LSB) to the most significant one (MSB). The module **M** subtracts the byte position from the rank and then adds the result to the real address. Following the example shown above, it gets a value of three for the rank a value of two for the byte position: 3−2=1; 00000010_{16}+1=00000011_{16}.

Figure 13.23. Module **M**.

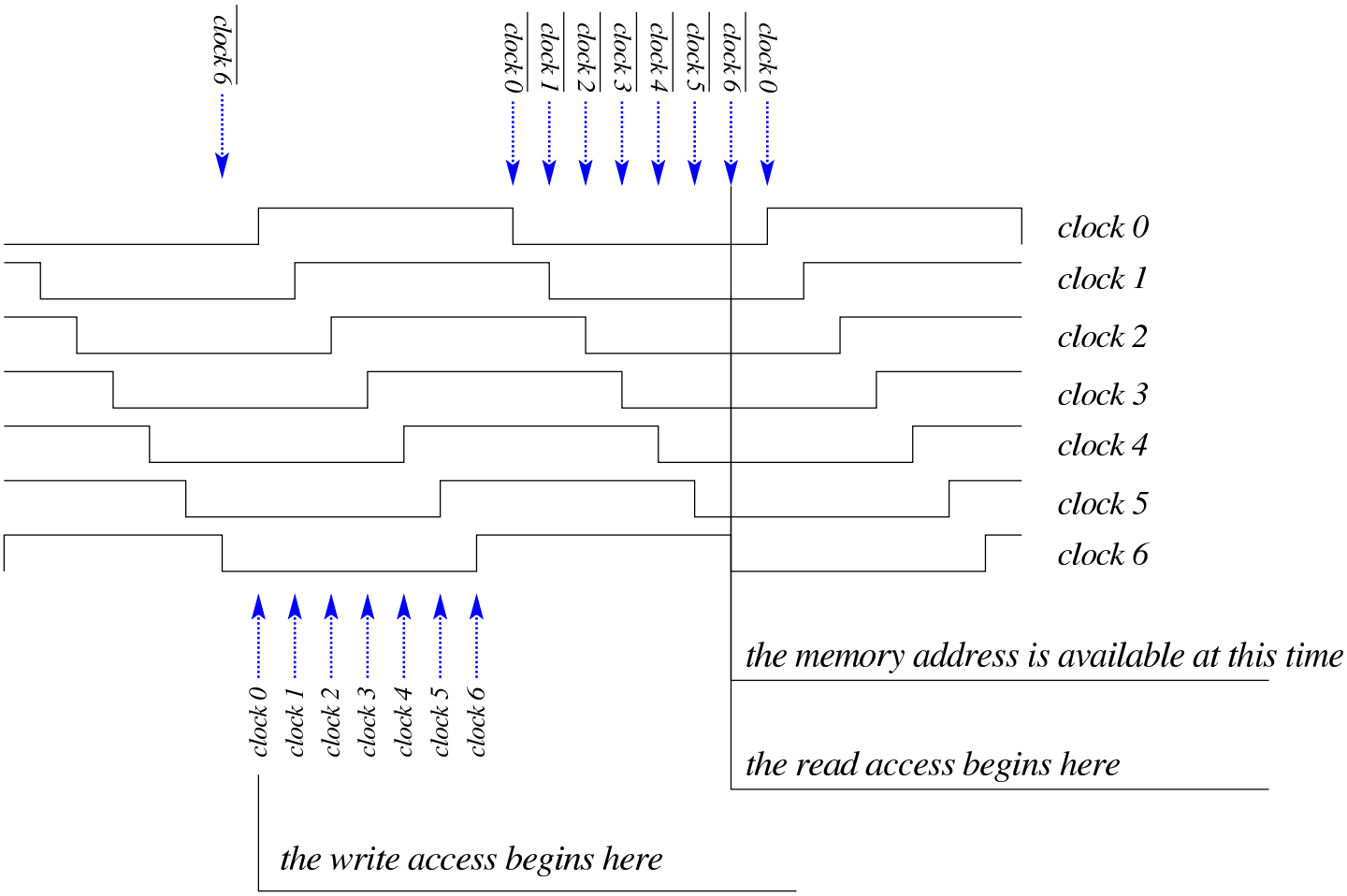


The calculated address is stored inside a 32-bit D flip-flop at the ***clock6*** edge:



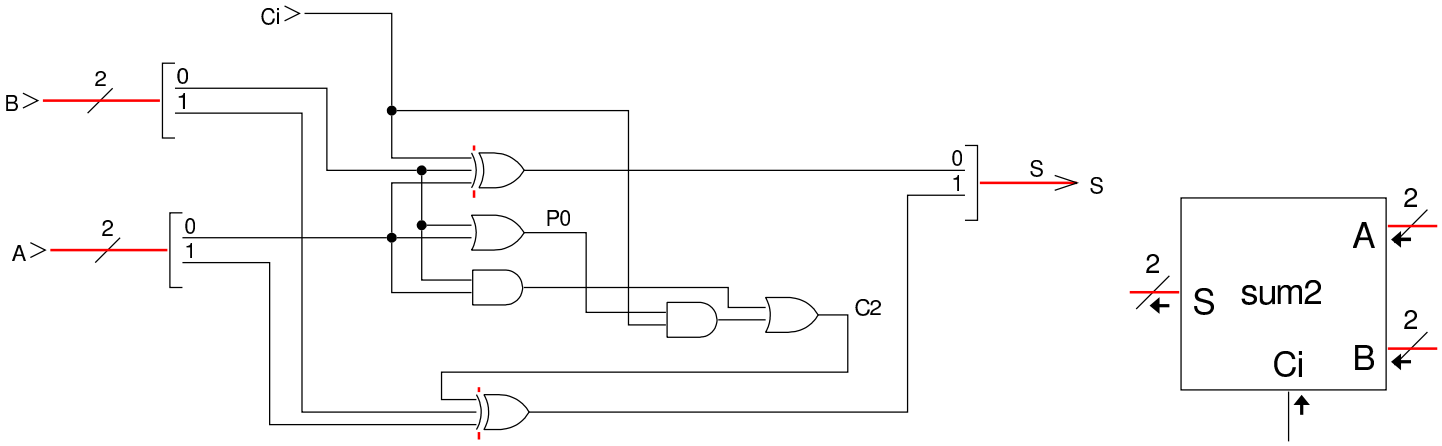
The memory is read starting from the ***clock6*** edge and is written starting from the ***clock0*** edge:

Figure 13.25. Read and write access timing.



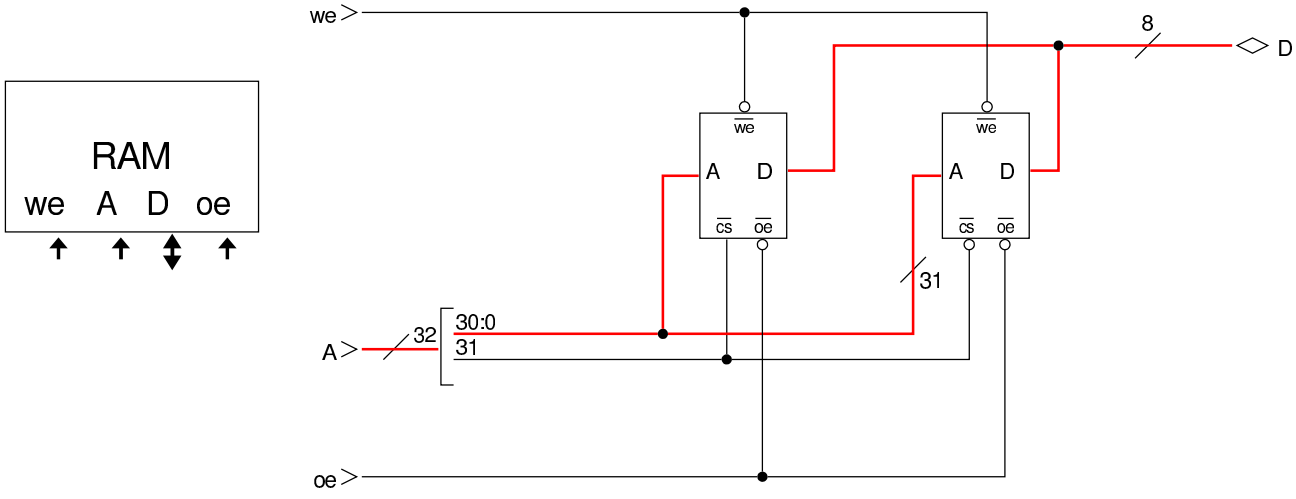
When something is read from the memory, the output is collected by the modules **MD** (*memory data*) or **IR** (*instruction register*). It depends on the type of data selected: the code is to be received by the ***IR*** register, whereas the other data is to be received by the ***MD*** register. When something is to be written to the memory, the byte is taken directly from the ***U*** bus, selecting the byte that is to be written. The module **M** has also an additional output ***IN*** that is used to connect the module **IVT**: this is done to emphasize the transition of a software interrupt number, read from the memory and received by the **IVT** module, it into a pointer to the IVT table in memory.

Figure 13.26. Module **sum2**, used to calculate the byte offset from the requested address, knowing the byte rank and the byte relative position.



The module **RAM** contains the RAM memory, to hide the fact that it is made of two memory components, because the TKGate version used to build and simulate this project is not able to handle memory components with more than 31 bits for the address.

Figure 13.27. Module **RAM**.



13.10 Module "IR"

The module **IR** is an 8-bit register —similar to the other registers— that is embedded inside the module **M**. The ***IR*** register never writes to the ***U*** bus and only a single line is used to control the register-load action. The register ***IR*** is updated at the ***clock2*** edge and its content is sent directly to the **CTRL** module.

Figure 13.28. The register ***IR*** is updated at the ***clock2*** edge.

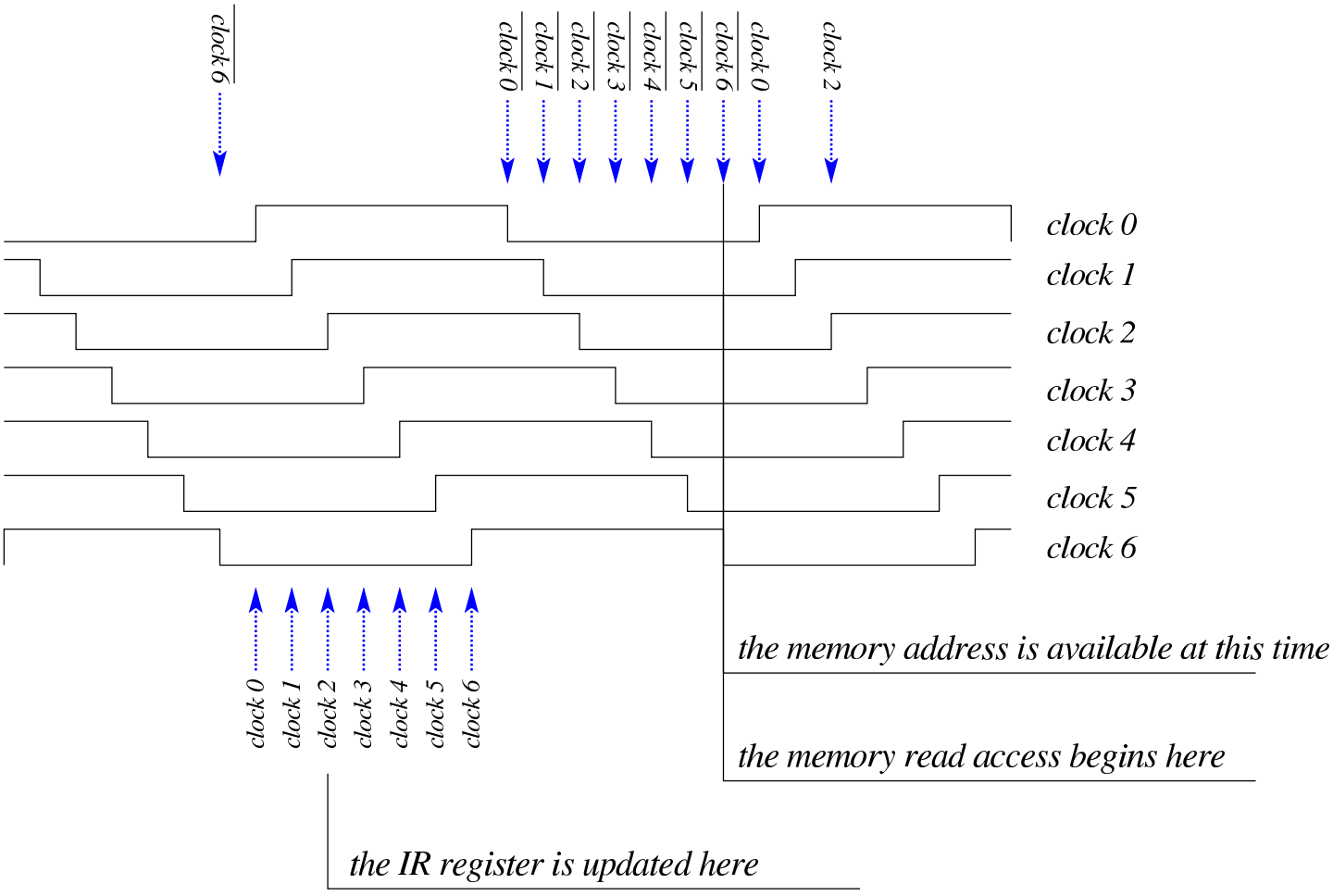
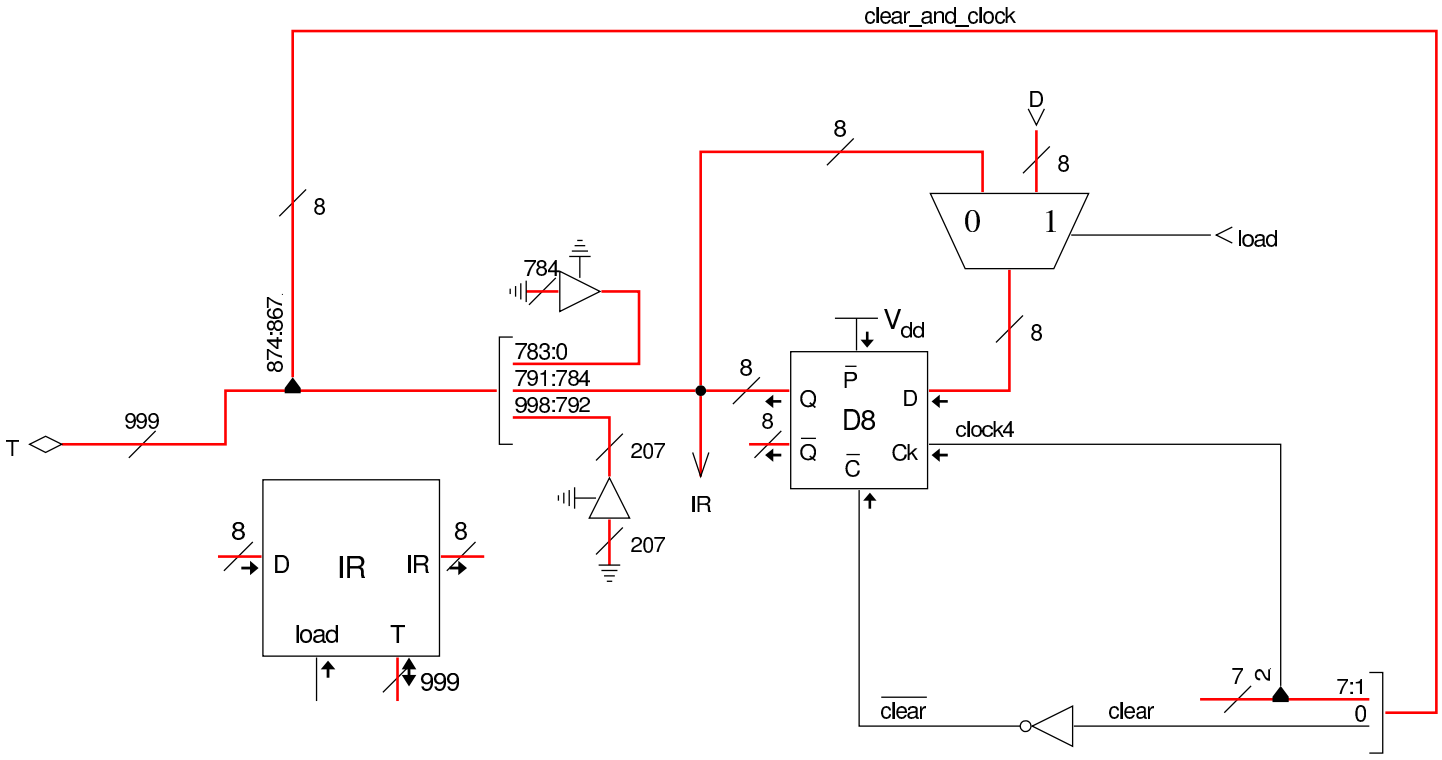


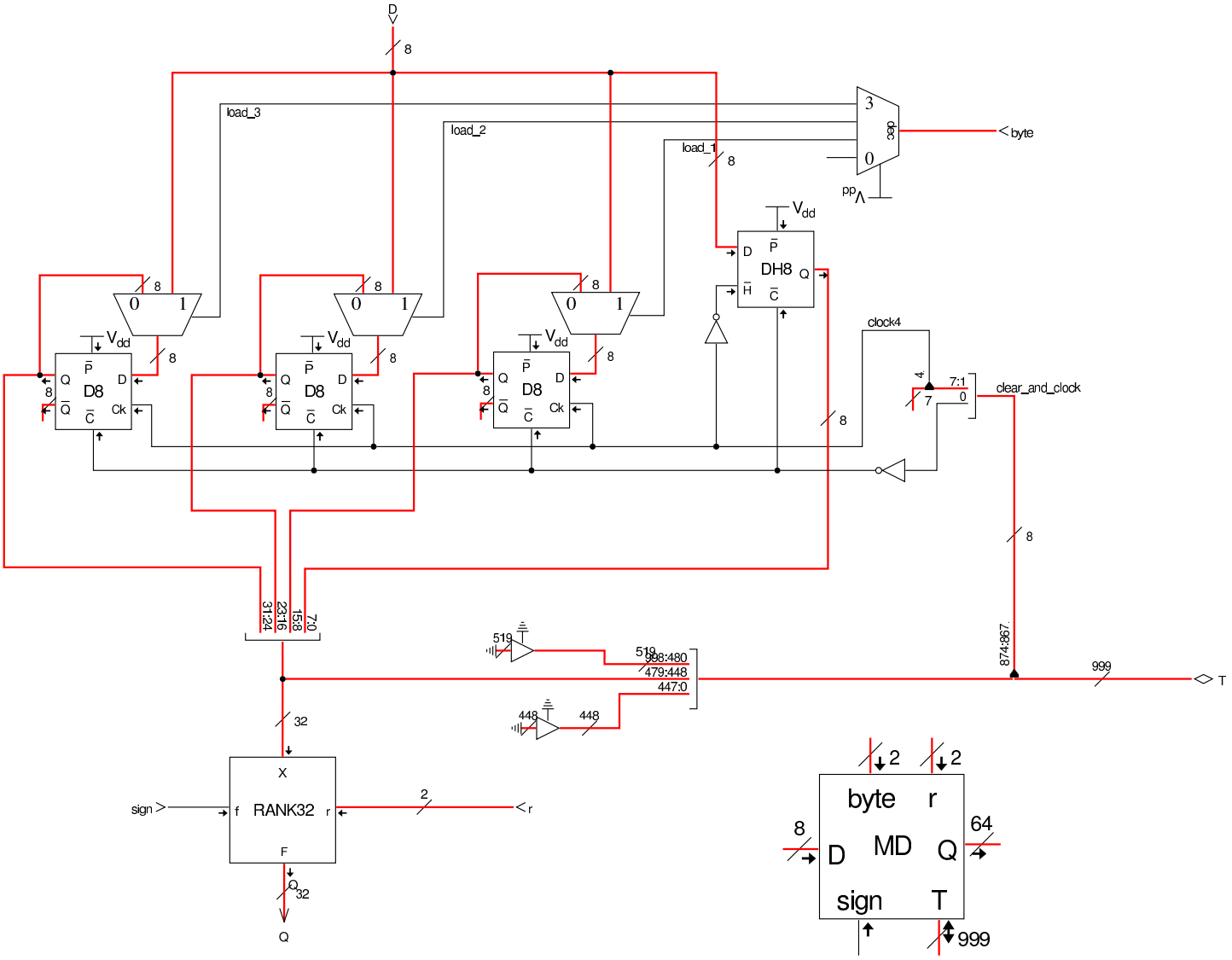
Figure 13.29. The ***IR*** register.



13.11 Module "MD"

The module **MD** is a register used to store a word, while it is read from the memory, byte after byte. The module requires to know which byte is readed from the expected word that is to be obtained. For example, when reading the byte two of a 32-bit word, the second 8-bit D flip-flop on the left is used to store the data.

Figure 13.30. Module **MD**.



The first 8-bit register on the right is different, because it is used a D latch instead, so that it holds the data when the clock is high, but the data received is already available on the output. The read access to the memory is done starting from the most significant byte, ending to the least significant one. That is: when reading the least significant byte all the other bytes are already available inside the other D flip-flop and the whole word is usually written to the ***U*** bus.

When writing to the ***U*** bus, the module **MD** requires to know the rank and if the value is signed or unsigned, so that the module **RANK32** is able to fix the final value is sent to the bus.

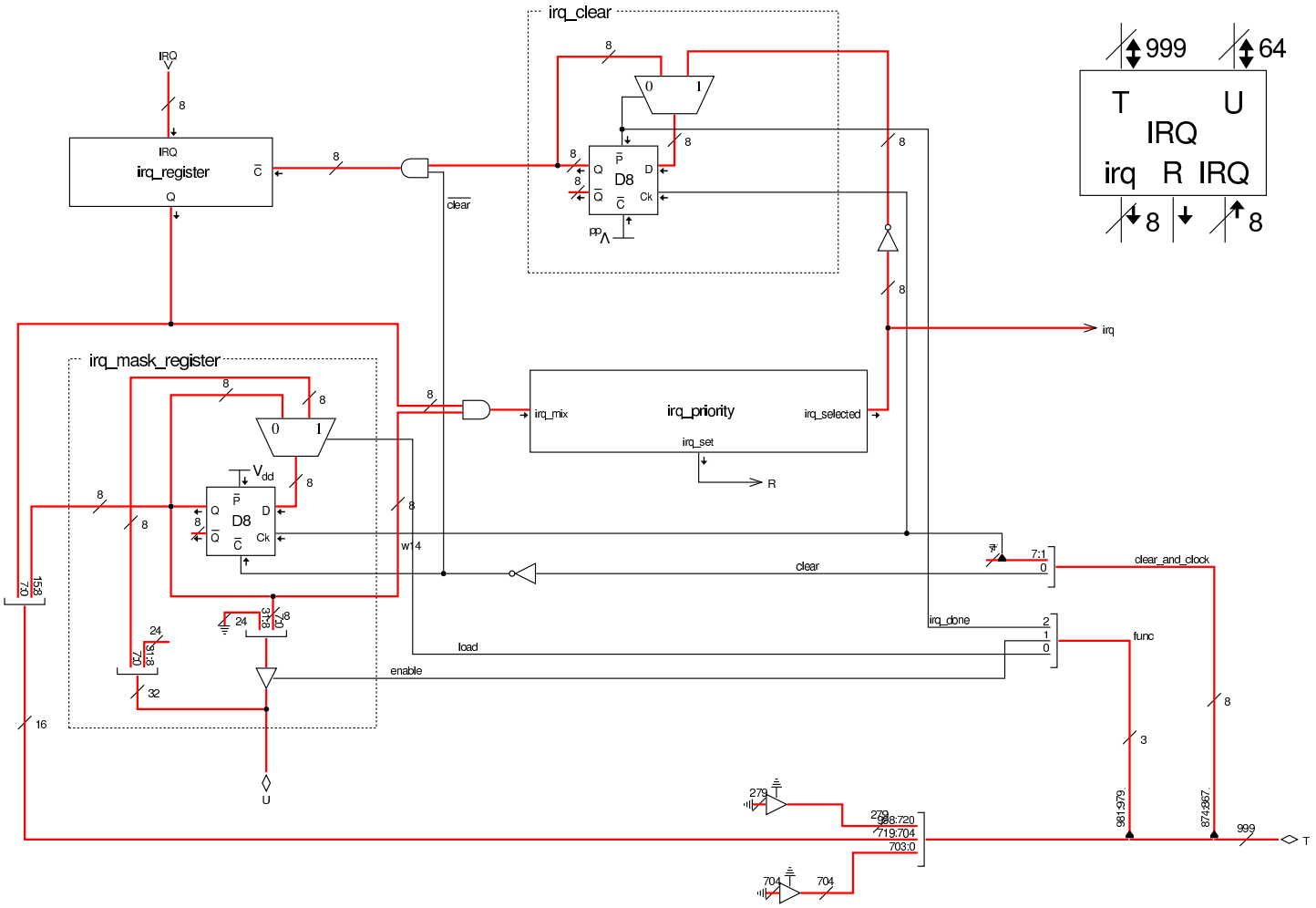
The clock signal used to update and hold the data received is ***clock4***, just like the other registers, but should be clear that when writing the ***U*** bus, the data is ready before the

clock4 edge, because the least significant byte is holded by the 8-bit D latch on the right.

13.12 Module "IRQ"

The module **IRQ** is the same as the previous version, except that it is able to handle eight different signals.

Figure 13.31. Module **IRQ**.



Another difference from the previous version of the module **IRQ** is that the selected IRQ (from the priority selection) is sent without conversions to the **IVT** module (it is the **IVT** module that converts what is necessary).

Figure 13.32. Module **`irq_priority`**.

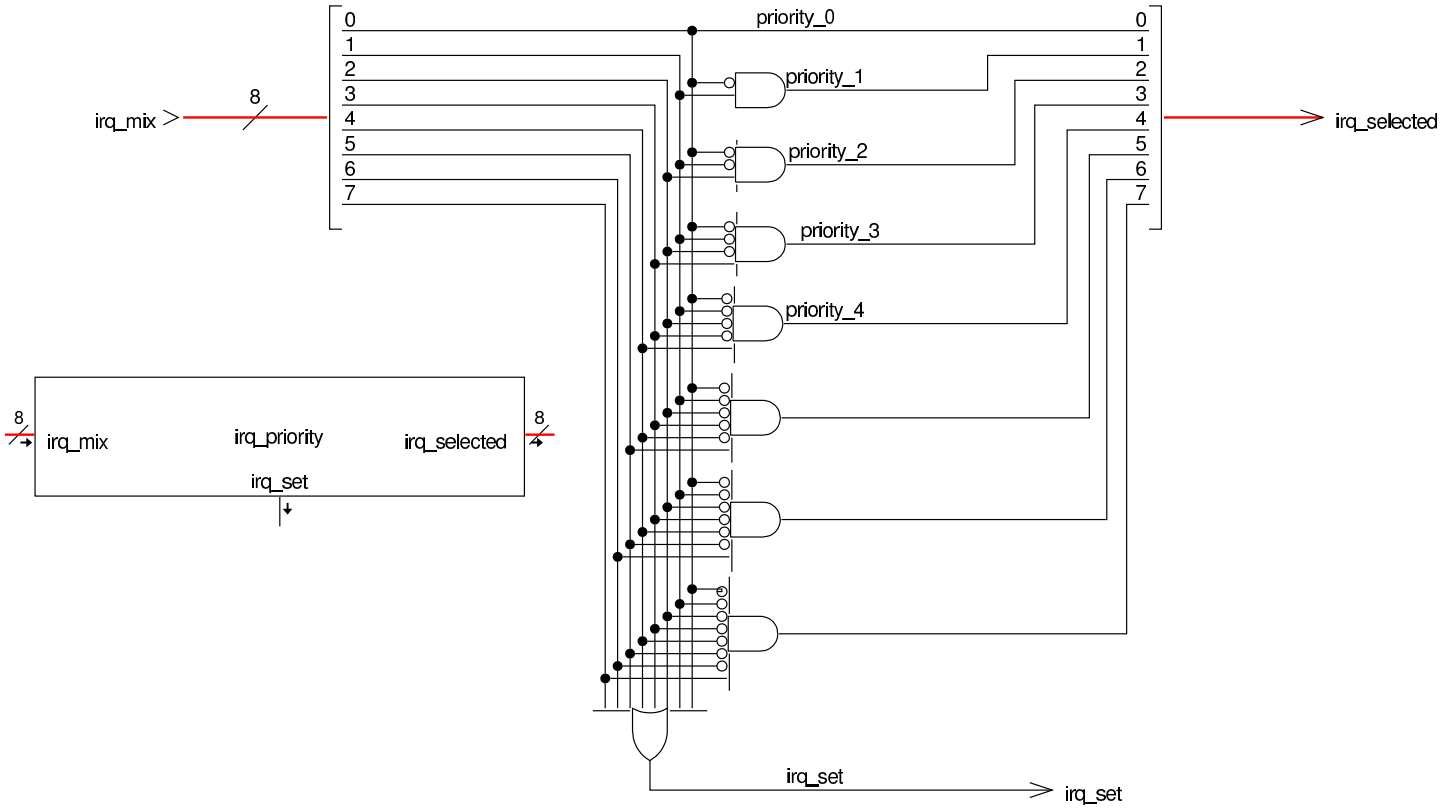
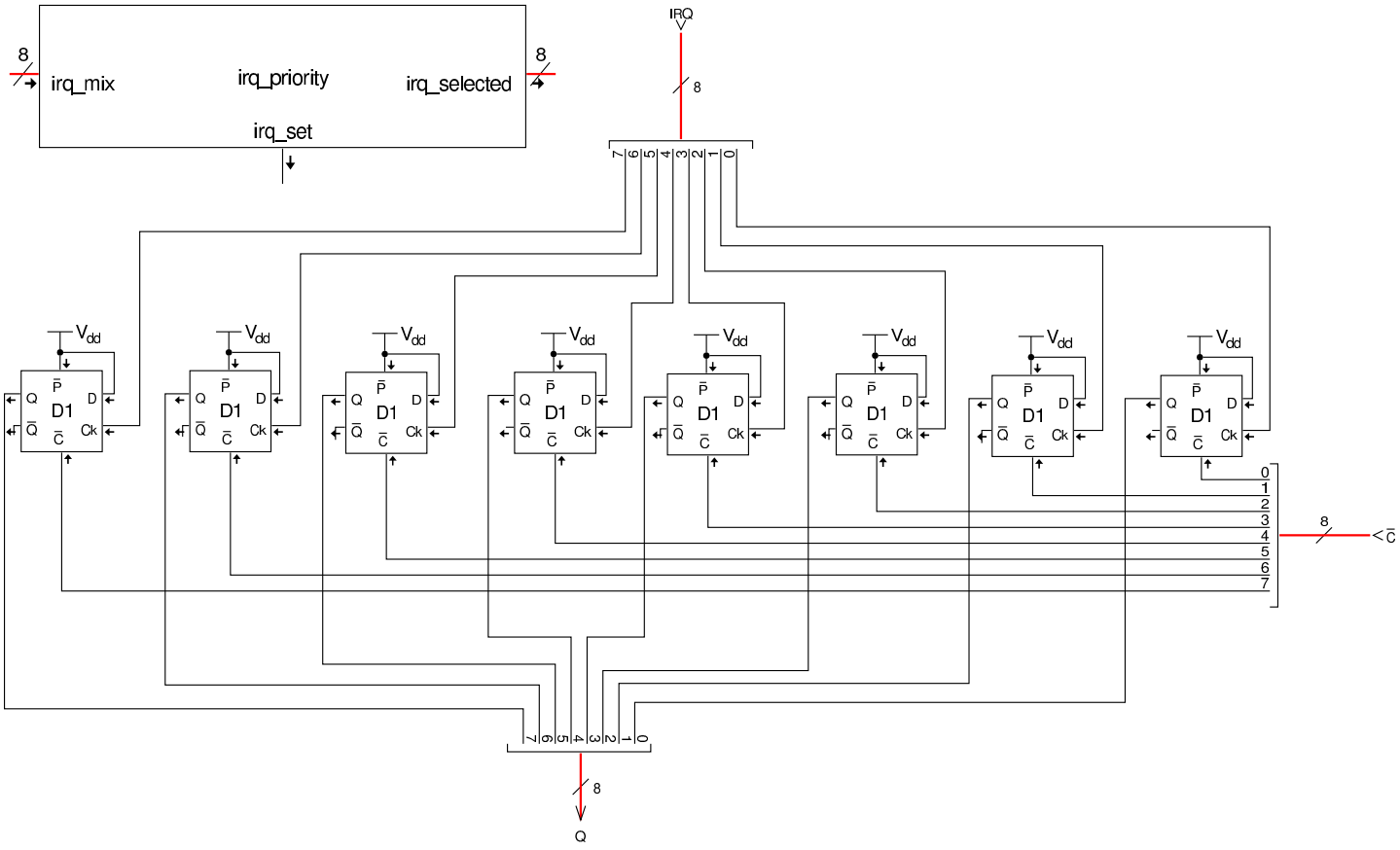


Figure 13.33. Module **`irq_register`**.

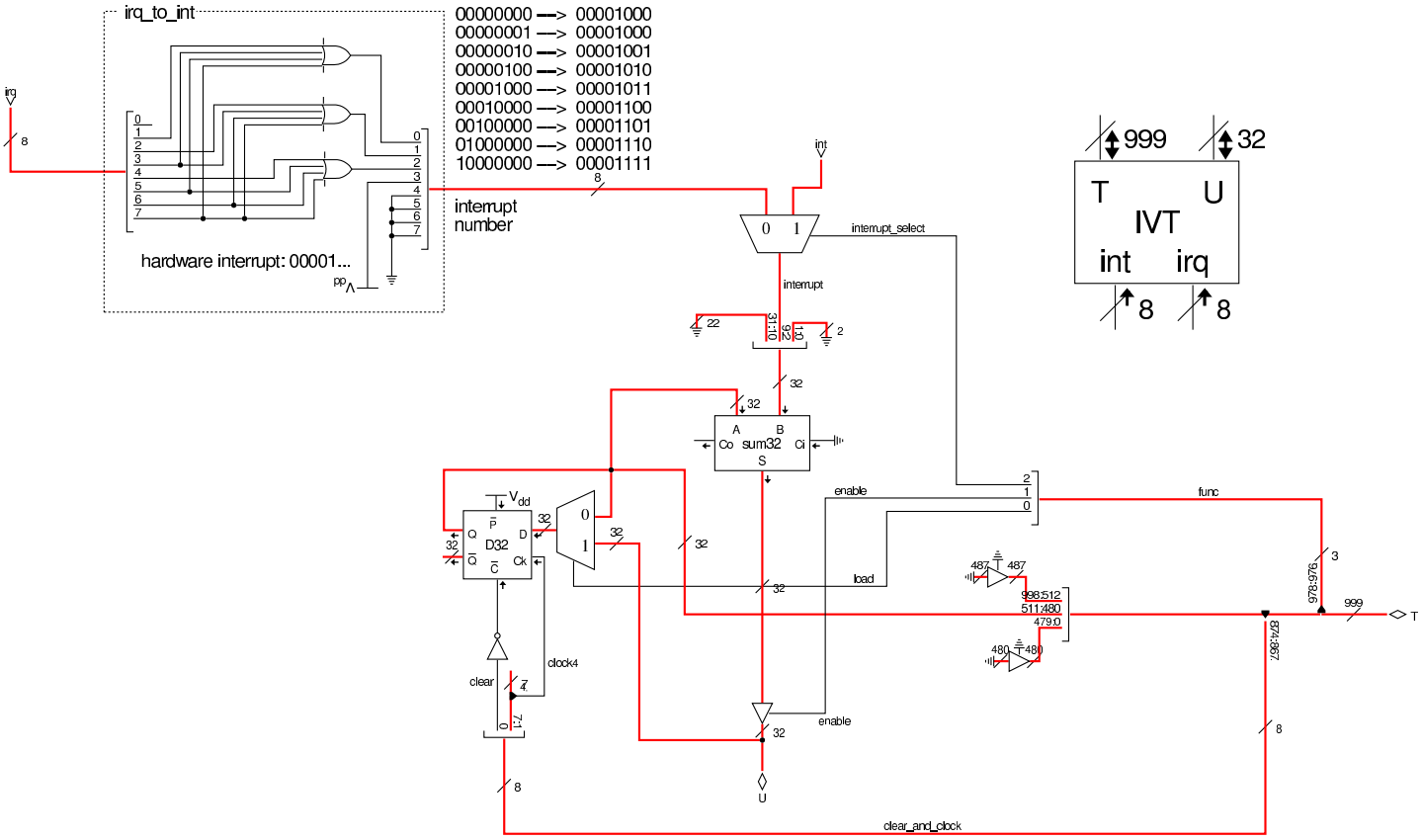


13.13 Module "IVT"

The **`IVT`** module is used to store the IVT (*interrupt vector table*) location and to calculate an address inside that table, when receiving an interrupt. The IVT table address is stored inside the 32-bit D flip-flop on the left; when receiving an interrupt from the ***int*** input line, or from the ***irq*** input line, the interrupt number is multiplied by four and added to the IVT table address: the resulting address available on the ***U*** bus, if writing on it is enabled, corresponds to the element of the IVT table that should contain a function address to process the interrupt received.

The interrupt is multiplied by four because each memory address occupies 32 bits (four bytes), consequently, every element of the IVT table is 32-bit long.

Figure 13.34. Module **IVT**.



It was decided to convert the IRQ numbers into interrupt numbers, adding the fixed value eight (1000_2). That is: IRQ 0 becomes INT 8, IRQ 1 becomes INT 9,... IRQ 7 becomes INT 15.

13.14 Module "ALU"

«

The module **ALU** is similar to the previous version, with the exception that 32 bits are used. Every operation that results on a value to be written in the ***U*** bus requires to specify the rank and if it is signed or not: the module **RANK32** would do the necessary adaptations and the flags carry, negative and overflow would be update consequently.

The ALU includes the module **SR32** that should be used to get a zero or −1 result, but it is not much useful because there is already the module **CNS** that can do it better.

With the module **FLSR** the ALU can modify the status register (***FL***), without hardware limitations: it is the microcode to establish if such modifications can be done.

Figure 13.35. Module **ALU**.

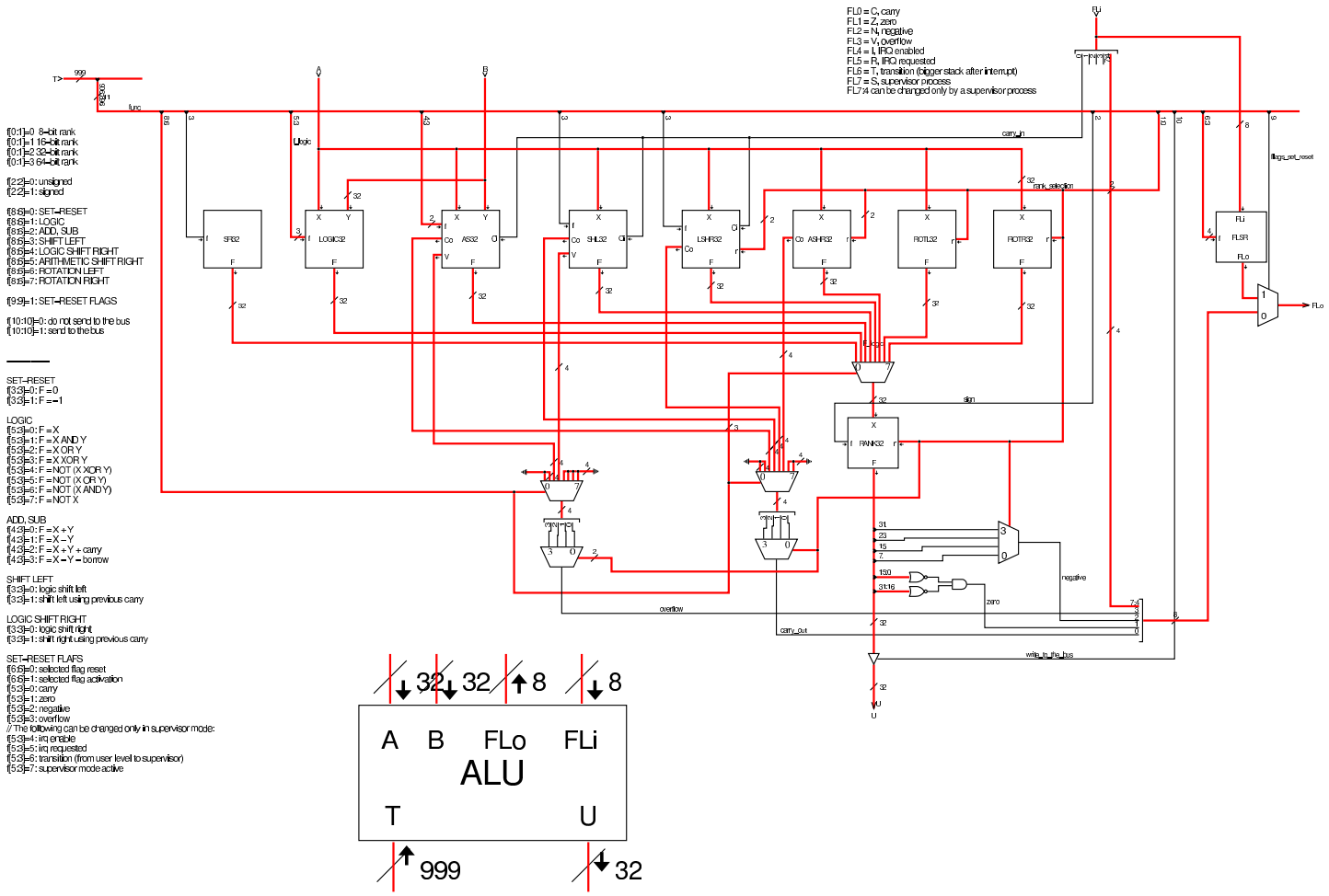


Figure 13.36. Module **SR**, used to generate the value zero or −1 (FFFFFFFF$_{16}$).

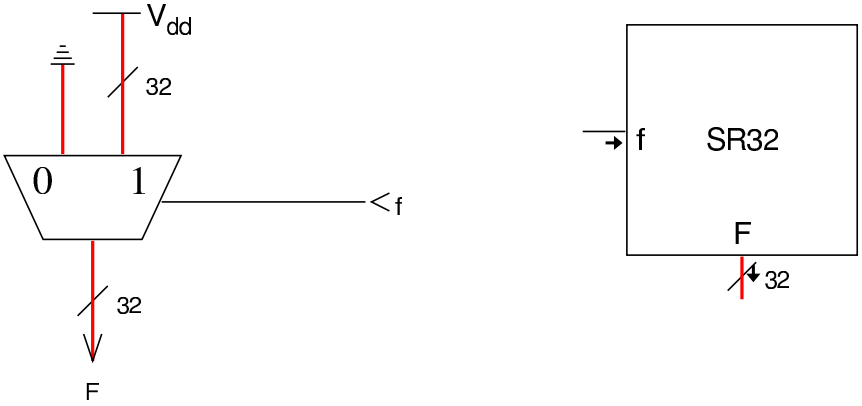
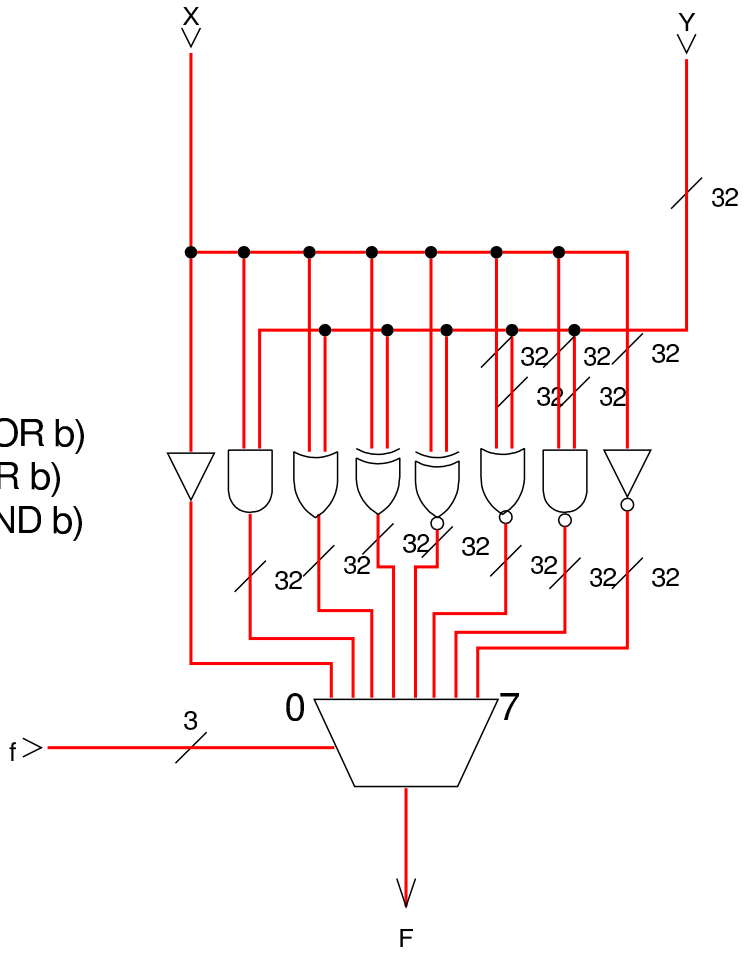


Figure 13.37. Module **LOGIC32**: bitwise boolean logic.

f=0: F = a
f=1: F = a AND B
f=2: F = a OR b
f=3: F = a XOR b
f=4: F = NOT (a XOR b)
f=5: F = NOT (a OR b)
f=6: F = NOT (a AND b)
f=7: F = NOT a



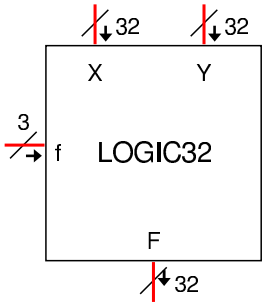


Figure 13.38. Module **AS32**: addition and subtraction.

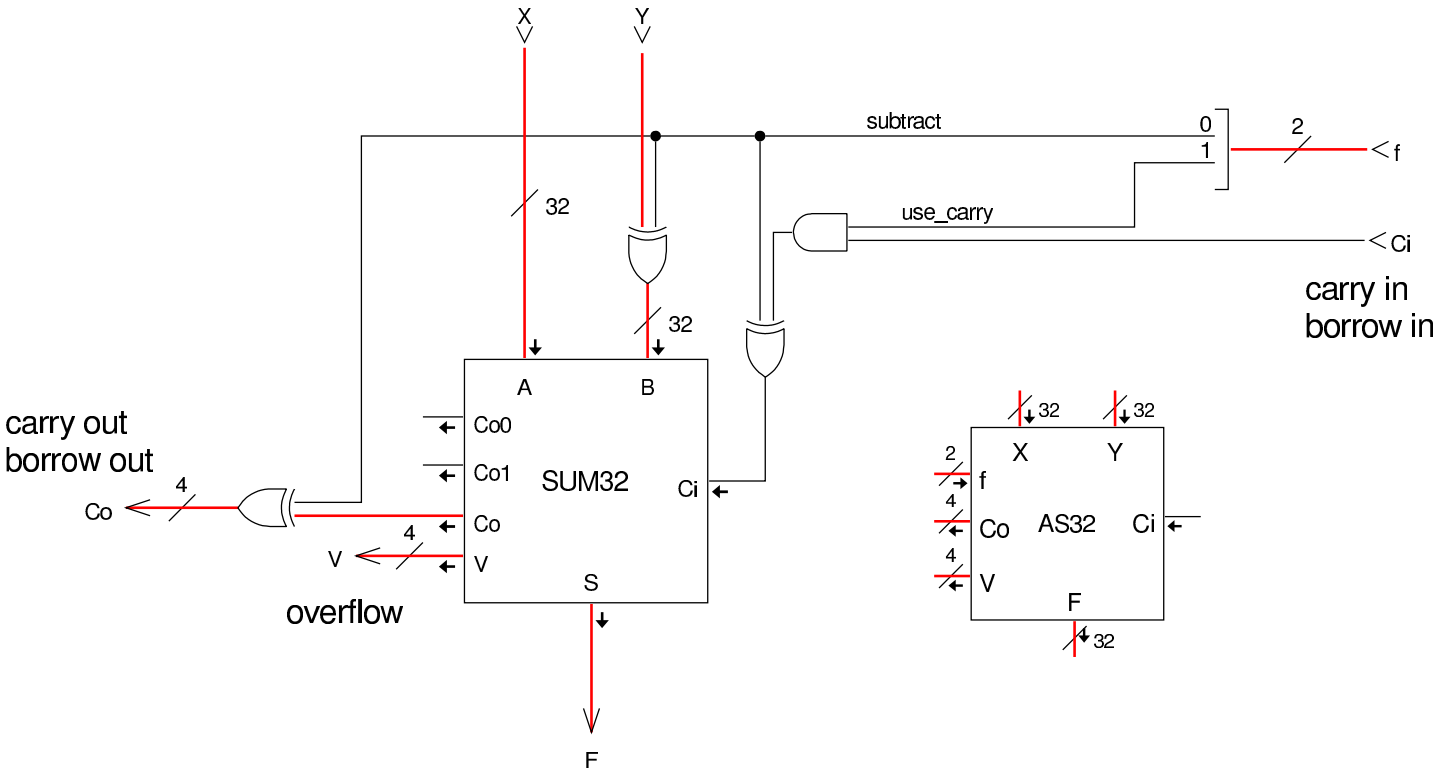


Figure 13.39. Module **SHL32**: any kind of binary shift left.

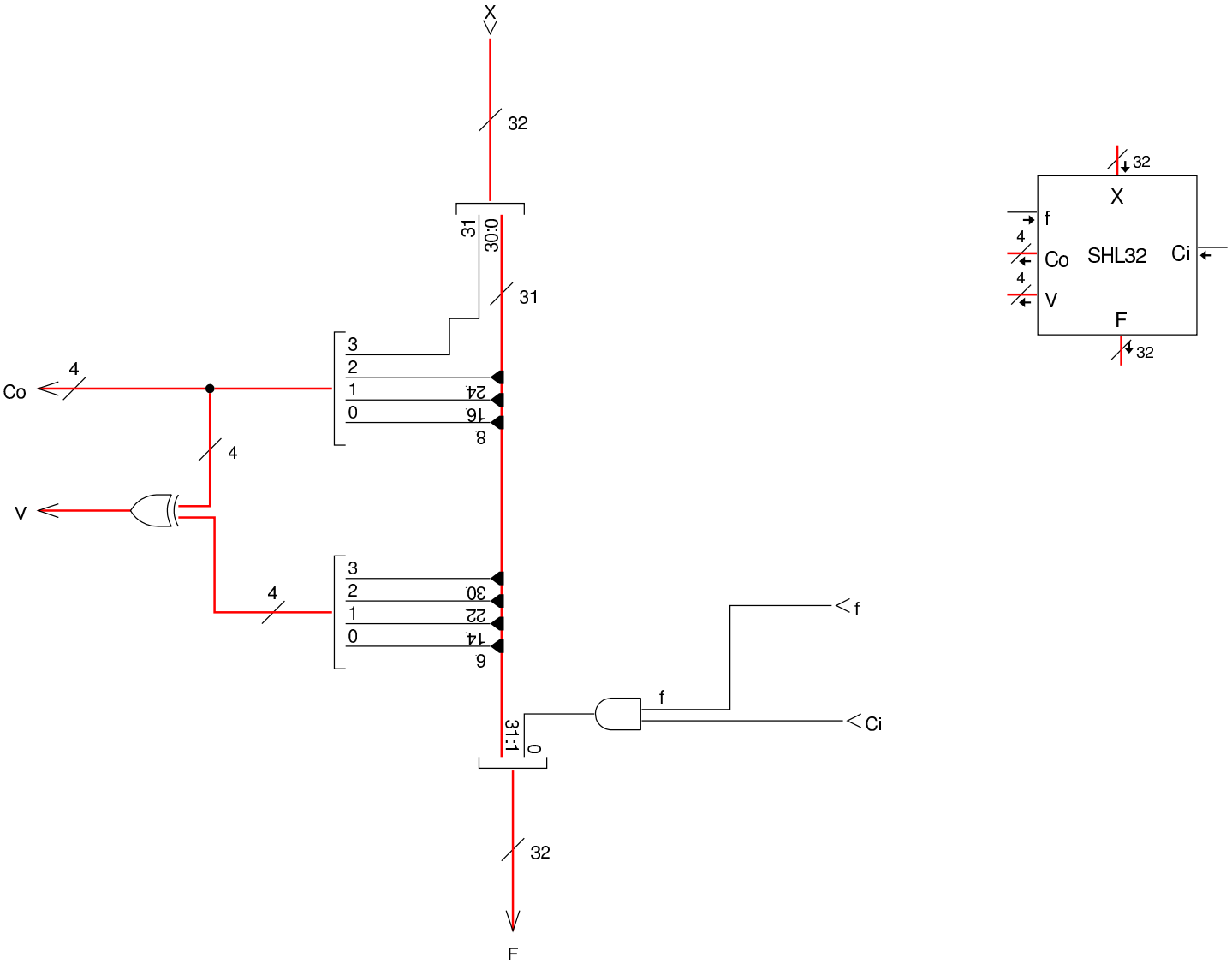


Figure 13.40. Module **LSHR32**: logic shift right.

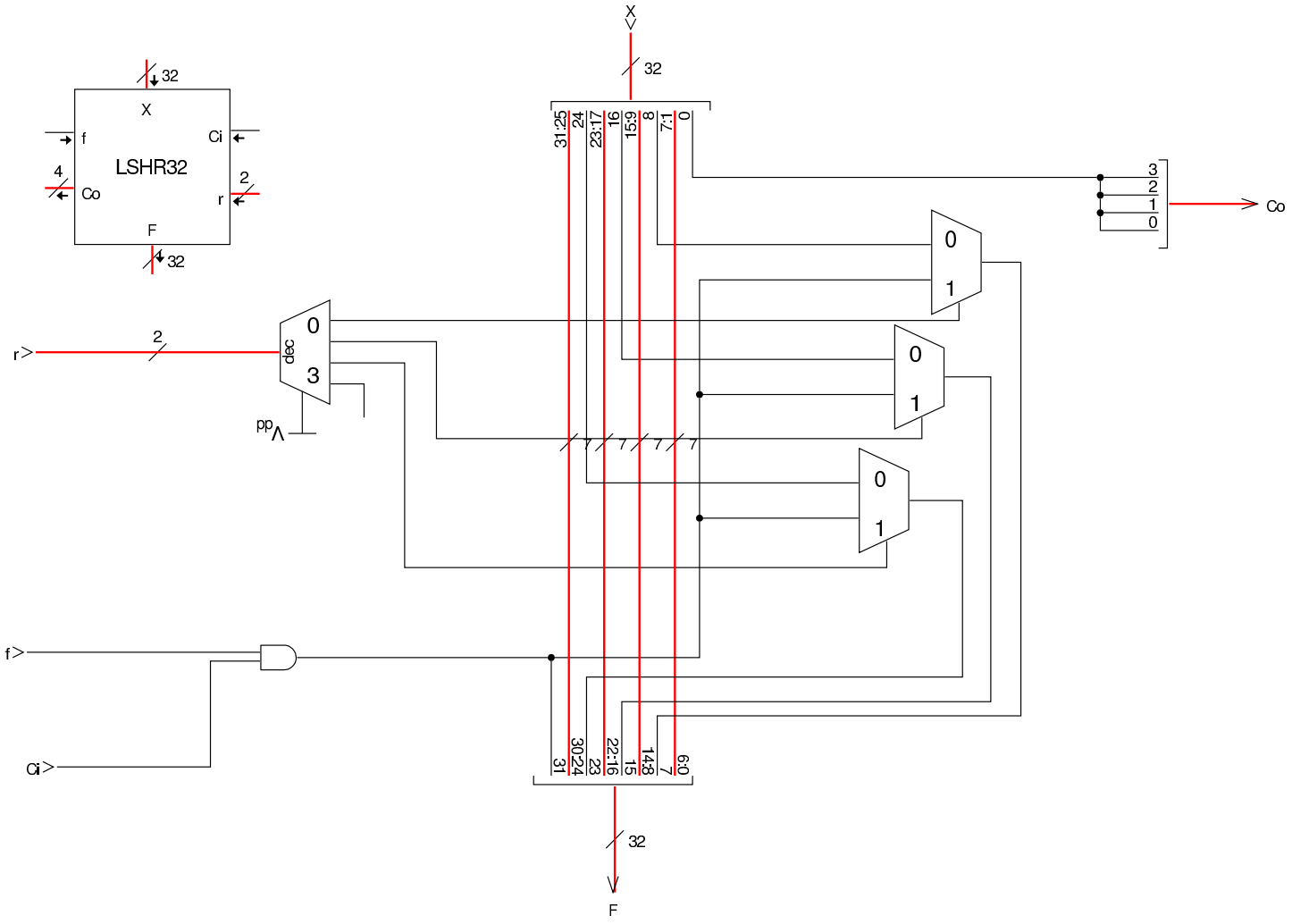


Figure 13.41. Module **ASHR32**: arithmetic shift right.

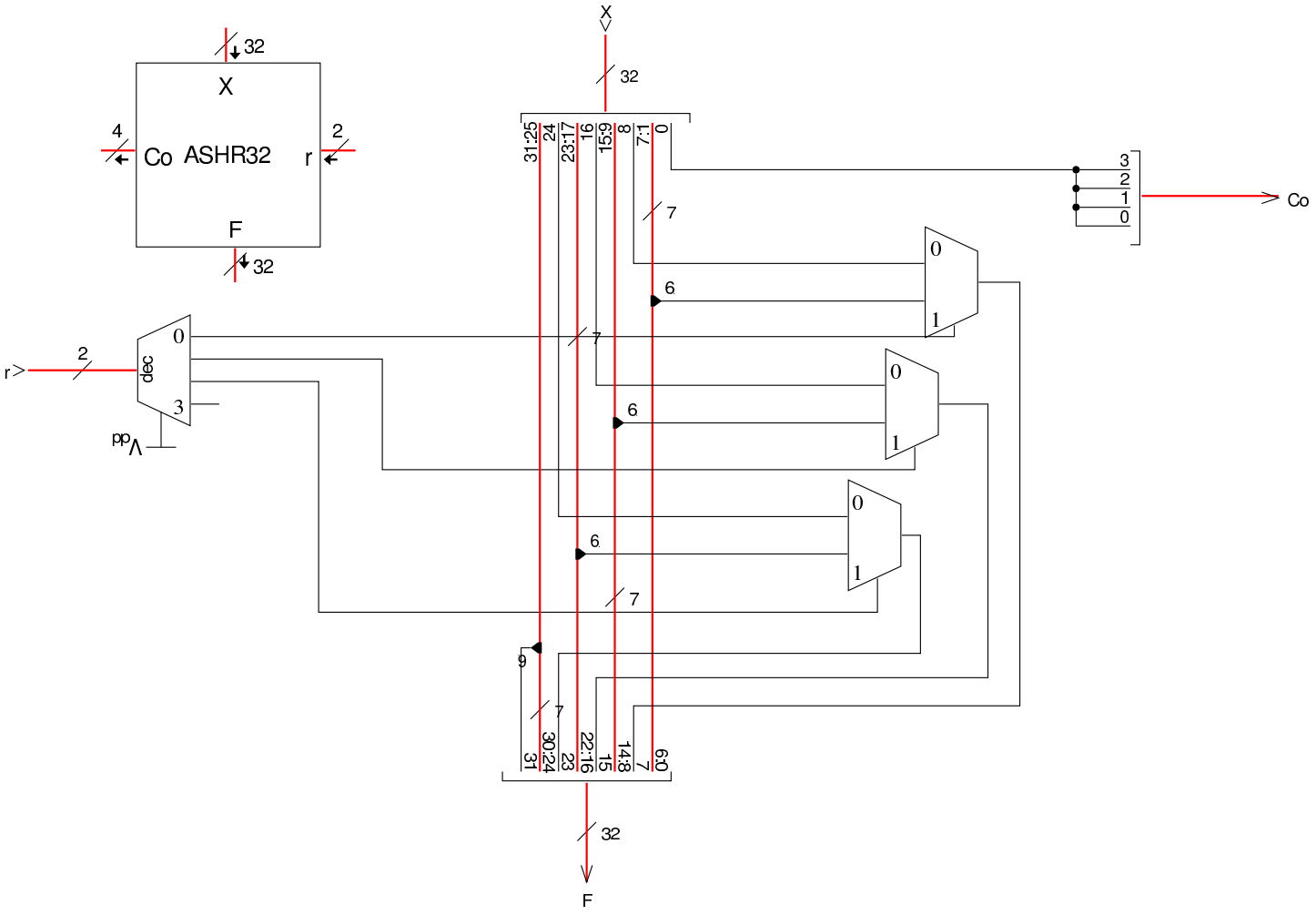


Figure 13.42. Module **ROTL32**: rotation left.

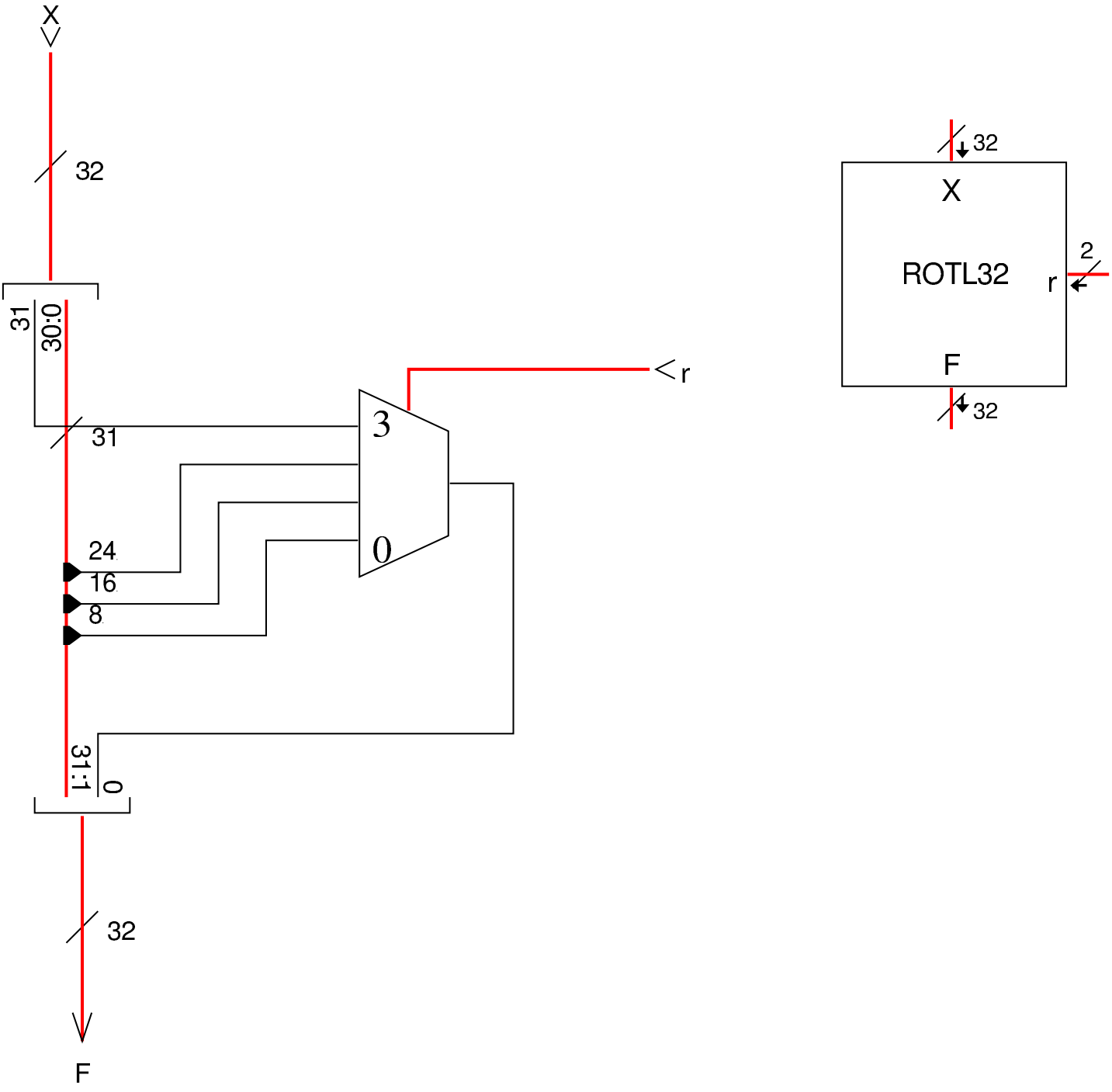


Figure 13.43. Module **ROTR32**: rotation right.

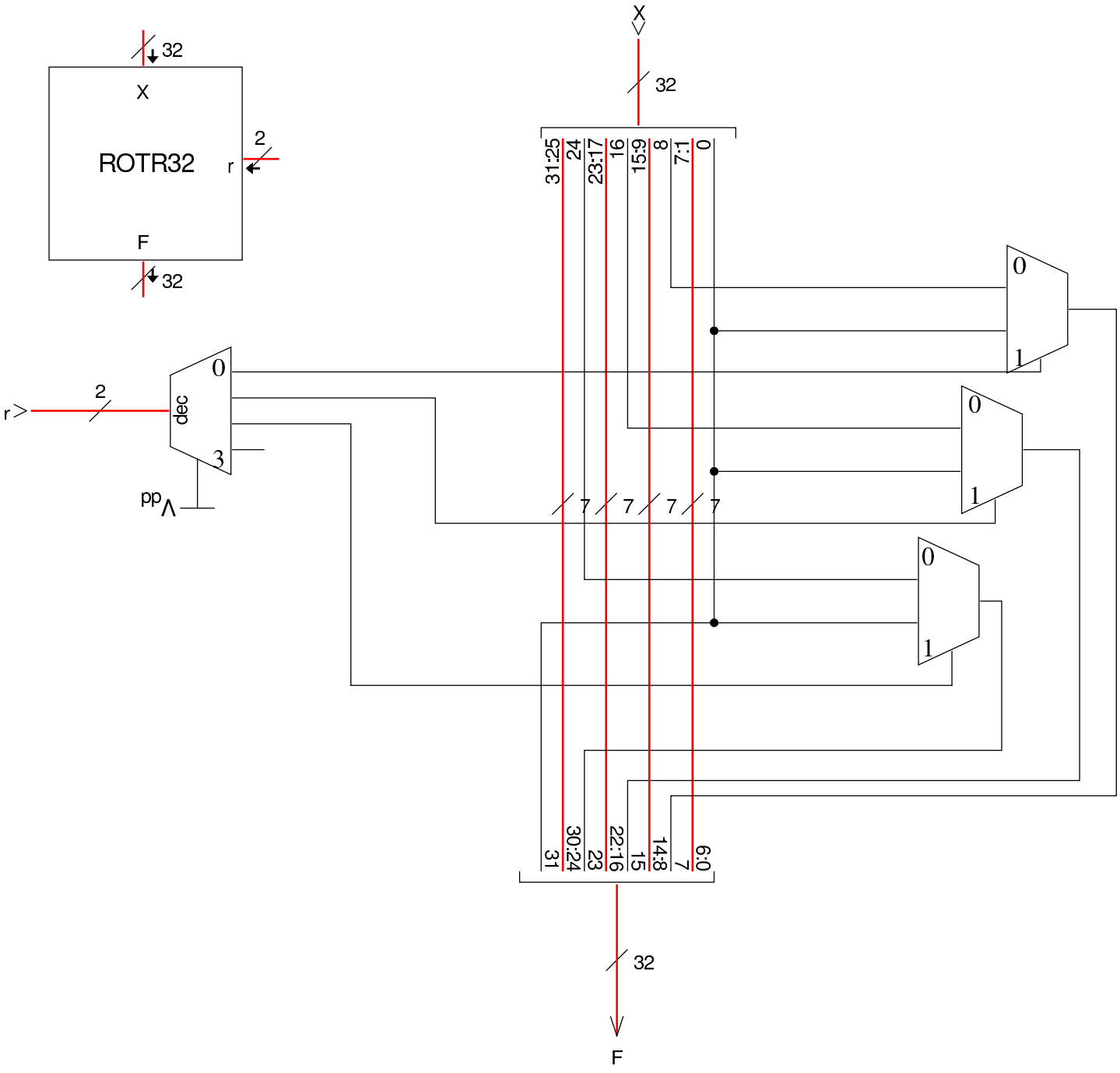
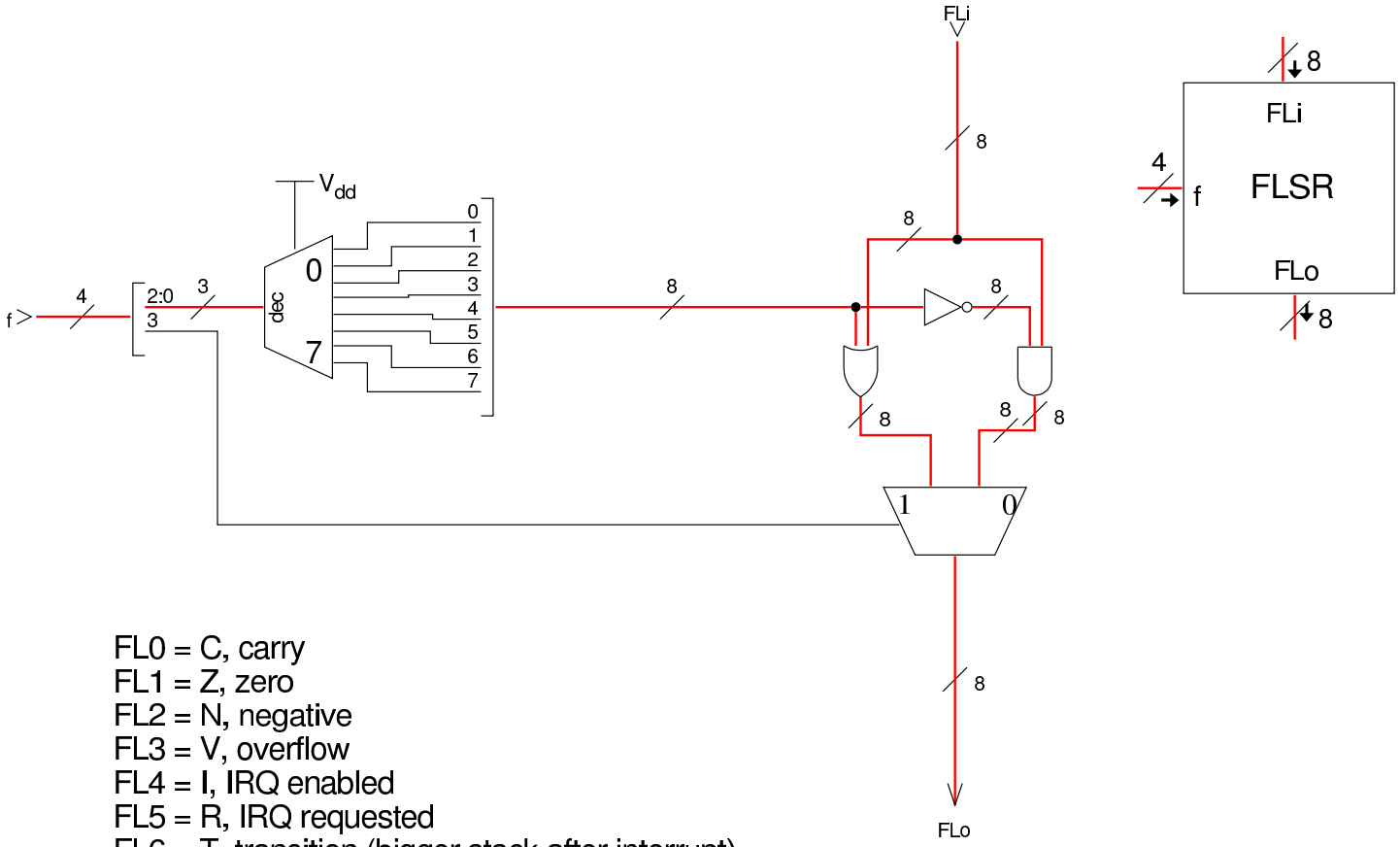


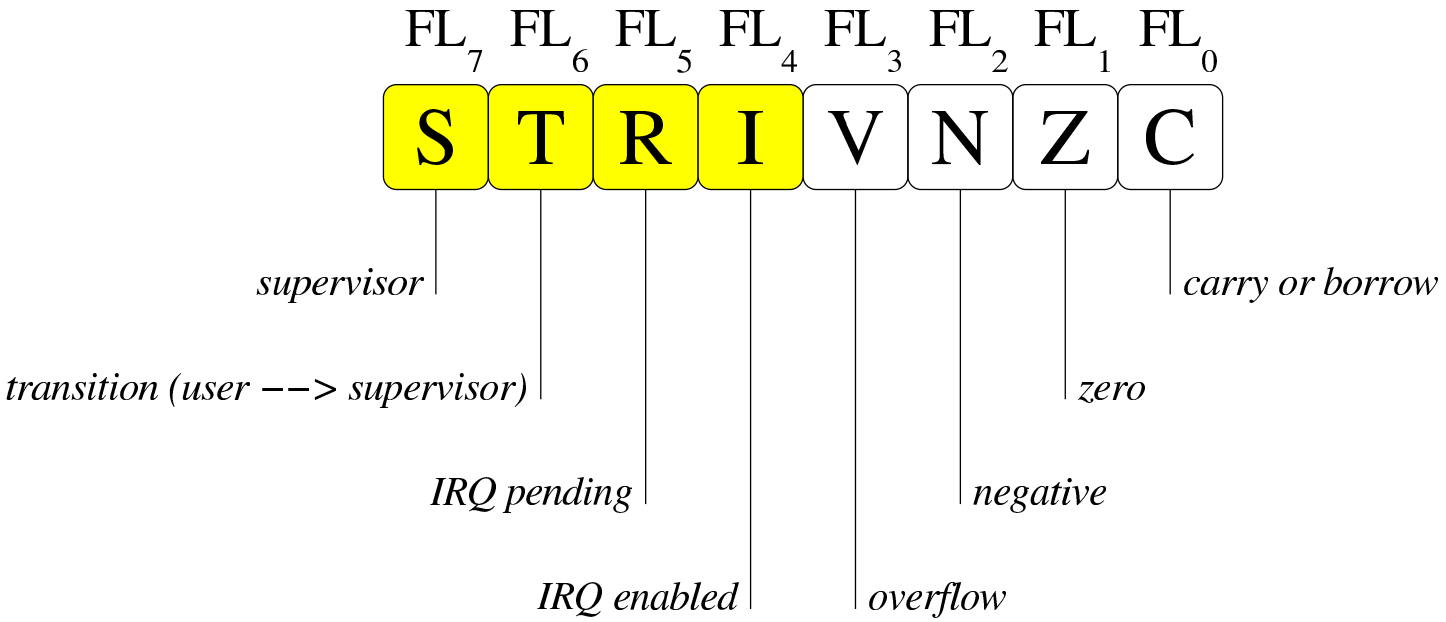
Figure 13.44. Module **FLSR**, used to change the flags inside the ***FL*** register. The module **FLSR** allows to change any flags, but it is checked inside the microcode if it is allowed.



FL0 = C, carry
FL1 = Z, zero
FL2 = N, negative
FL3 = V, overflow
FL4 = I, IRQ enabled
FL5 = R, IRQ requested
FL6 = T, transition (bigger stack after interrupt)
FL7 = S, supervisor process
FL7:4 can be changed only by a supervisor process (as defined inside the microcode)

13.15 Module "FL"

Flags are more sophisticated inside this version of the simple CPU, because half of the flags can be modified only by a supervisor process. Here are the current flags meaning:



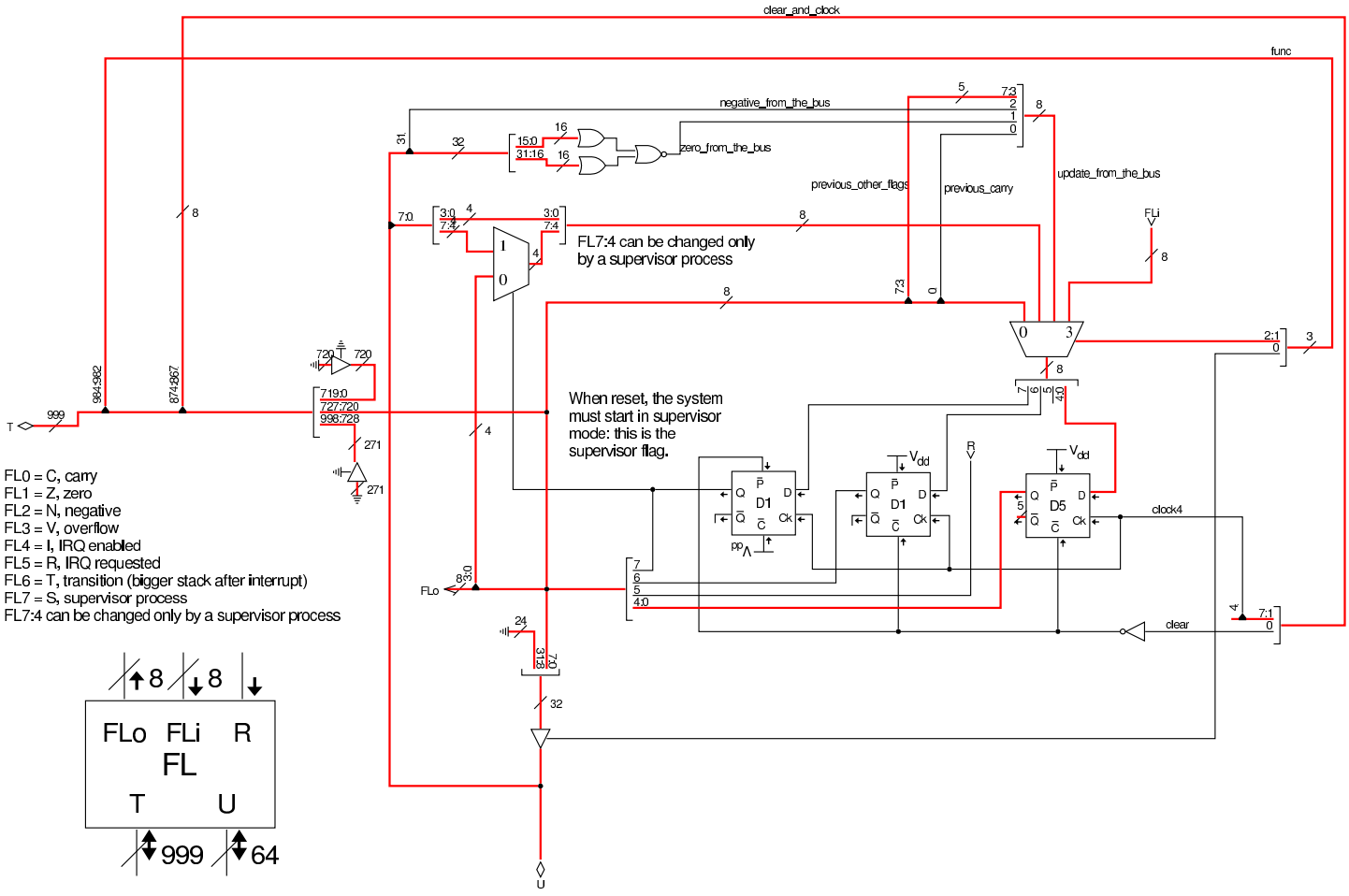
The figure shows the flags $\boldsymbol{FL}_{7:4}$ emphasized, because these flags can be modified only when the 'supervisor' flag is already active. A supervisor process can decide to lose its own privileges, but then it cannot gain them again, except when an interrupt is processed: it is supposed that the interrupt procedures listed inside the IVT table are sure enough to be executed with privileges.

The ***FL*** register is different from the previous version, because it should not allow to gain privileges when not allowed. The register can be loaded form four the ***U*** or from the **`ALU`** module, but when loading from the bus, it is possible to chose to update only the zero and negative flags. In other words, the ***FL*** can: keep the value stored, load a new value from the bus, update the zero and negative flags from the value available in the bus, load a new value from the ALU.

When loading a new value from the bus, the register can accept to modify the bits $\boldsymbol{FL}_{7:4}$ only if the supervisor flag is already active.

The bit $\boldsymbol{FL}_5$ is never stored, because it just comes from the **`IRQ`** module and it represent a pending IRQ request, which might or might not be processed (it depends on the flag $\boldsymbol{FL}_4$).

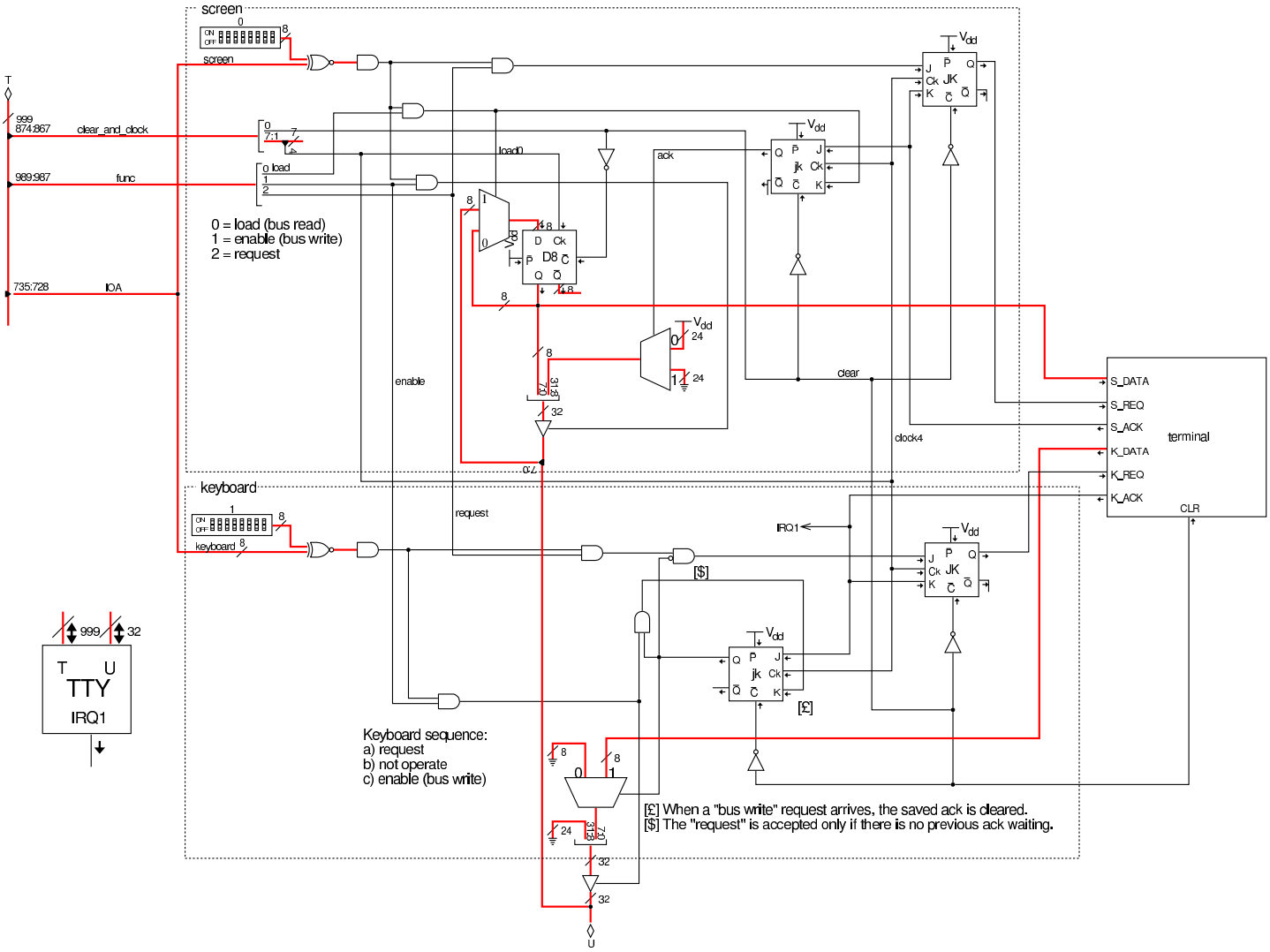
Figure 13.46. Module **FL**.



13.16 Module “TTY”

The **TTY** module is almost the same as the previous version, replacing the old module **DR8** with a D flip-flop with external multiplexer. The module **TTY** is a textual screen-keyboard terminal interface. All the details about the module functionality should be found at some previous versions description.

Figure 13.47. Module **TTY**.



Listing 13.48. Module **terminal**: Verilog code.

```
module terminal(K_DATA, K_REQ, K_ACK, S_DATA, S_REQ, S_ACK, CLR);

output K_ACK;
output S_ACK;
output [7:0] K_DATA;
input [7:0] S_DATA;
```

```
input K_REQ;
input S_REQ;
input CLR;
reg k_ready;
reg [7:0] key;
reg s_ready;

  initial
    begin
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  always
    begin
      @(posedge CLR)
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  initial $tkg$post("TERMINAL","%m");

  always
    begin
      @ (posedge K_REQ);
      # 5;
      key = $tkg$recv("%m.KD");
      # 5;
      k_ready = 1'b1;
      # 5;
      @ (negedge K_REQ);
      # 5;
      k_ready = 1'b0;
    end
```

```
  always
    begin
      @(posedge S_REQ);
      # 5;
      $tkg$send("%m.SD",S_DATA);
      # 5;
      s_ready = 1'b1;
      # 5;
      @(negedge S_REQ);
      # 5;
      s_ready = 1'b0;
    end

  assign S_ACK = s_ready;
  assign K_DATA = key;
  assign K_ACK = k_ready;

endmodule
```

Listing 13.49. File '`share/tkgate/vpd/terminal.tcl`' for TCL interface.

```
image create bitmap txtcurs -file "$bd/txtcurs.b"

VPD::register TERMINAL
VPD::allow TERMINAL::post
VPD::allow TERMINAL::data

namespace eval TERMINAL {
    # Public variables declarations: the variables $terminal_...
    # are arrays of which only the element $n is used;
    # that element identifies uniquely the working interface instance.
    variable terminal_w
    variable terminal_pos
    #
    variable KD
    # Function requested by TKGate to create the interface.
    proc post {n} {
        variable terminal_w
        variable terminal_pos
        # Create the window and save the object element in a $terminal_w array element.
        set terminal_w($n) [VPD::createWindow "TERMINAL $n" -shutdowncommand "TERMINAL::unpost $n"]
        # For convenience, copy the object reference inside the local
        # variable $w; then, the variable $w will be used as a reference to the object.
        set w $terminal_w($n)
```

```
        text $w.txt -state disabled
        pack $w.txt
        # Put the cursor at the end of the displayed text.
        $w.txt image create end -image txtcurs
        # Bind the keyboard input, related to the object represented by
        # $terminal_w($n), to the function sendChar.
        bind $w <KeyPress> "TERMINAL::sendChar $n \"%A\""
        # Open a reading channel, named «SD» (screen data),
        # and associate it to the function «data»; moreover, open a
        # writing channel, named «KD» (keyboard data).
        if {[info exists ::tkgate_isInitialized]} {
            VPD::outsignal $n.KD TERMINAL::KD($n)
            VPD::insignal $n.SD -command "TERMINAL::data $n" -format %d
        }
        # Reset the character count, used to count the characters displayed
        # on screen.
        set terminal_pos($n) 0
    }
    # Function that receives the typing and put it into the
    # channel «KD», related to the current interface instance.
    proc sendChar {n key} {
        variable KD
        if { [string length $key ] == 1 } {
            binary scan $key c c
            set TERMINAL::KD($n) $c
        }
    }
    # Function that TKGate requires to destroy the interface.
    proc unpost {n} {
        variable terminal_w
        variable terminal_pos
        destroy $terminal_w($n)
        destroy $terminal_pos($n)
        unset terminal_w($n)
        unset terminal_pos($n)
    }
    # Function used to get the data to display on screen.
    proc data {n c} {
        variable terminal_w
        variable terminal_pos
        # For convenience, copy the object reference representing the
        # interface, inside the variable $w.
        set w $terminal_w($n)
        catch {
            # The variable $c contains the character to display.
            if { $c == 7 } {
                # BEL
                bell
                return
            } elseif { $c == 127 || $c == 8 } {
                # DEL | BS
                if { $terminal_pos($n) > 0 } {
```

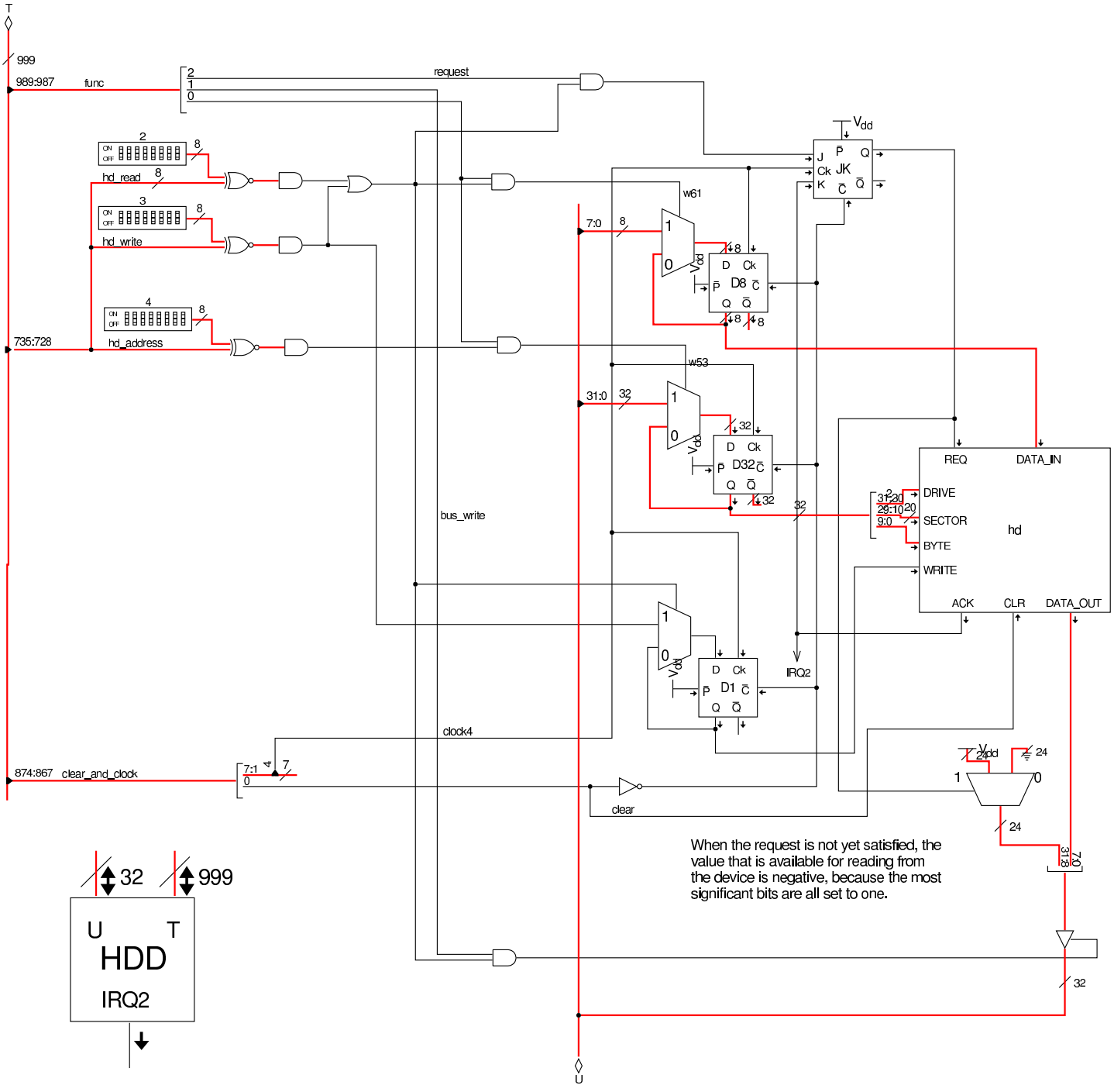
```
                    # Delete the last displayed character, but only if the
                    # characters counter is greater than zero, otherwise
                    # the cursor would disappear and the next characters
                    # would be located in an unvisible screen area.
                    $w.txt configure -state normal
                    $w.txt delete "end - 3 chars"
                    $w.txt see end
                    $w.txt configure -state disabled
                    set terminal_pos($n) [expr {$terminal_pos($n) - 1}]
                }
                return
            } elseif { $c == 13 } {
                # Convert CR to LF.
                set c 10
            }
            # Convert the character number into a visible symbol.
            set x [format %c $c]
            # Display the symbol.
            $w.txt configure -state normal
            $w.txt insert "end - 2 chars" $x
            $w.txt see end
            $w.txt configure -state disabled
            # Update the displayed characters counter.
            set terminal_pos($n) [expr {$terminal_pos($n) + 1}]
        }
    }
}
```

13.17 Module “HDD”

«

The **HDD** module similar to the previous version: the address is now specified with a single 32-bit request and only four virtual units are available. The **HDD** module. All the details about the module functionality should be found at some previous versions description.

Figure 13.50. Module **HDD**.



Listing 13.51. Verilog code describing the module **hd**.

```
module hd(DRIVE, SECTOR, BYTE, WRITE, DATA_IN, DATA_OUT, REQ, ACK, CLR);
input [1:0] DRIVE;
input WRITE, REQ, CLR;
input [19:0] SECTOR;
input [9:0] BYTE;
input [7:0] DATA_IN;
```

```
output [7:0] DATA_OUT;
output ACK;
//
integer _data_out;
integer _ack;
//
reg [7:0] buffer[0:1023];
reg [8*24-1:0] filename = "hd0_sector_000000000.mem";
//
integer i;
integer sector_8;
integer sector_7;
integer sector_6;
integer sector_5;
integer sector_4;
integer sector_3;
integer sector_2;
integer sector_1;
integer sector_0;
integer x;
//
  initial
    begin
      for (i=0; i<1024; i=i+1)
        begin
          //
          // Initial buffer reset with 00.
          //
          buffer[i] = 8'h00;
        end
      _ack = 0;
      _data_out = 0;
      x = 0;
    end
  //
  always
    begin
      @(posedge CLR)
      _ack = 0;
      _data_out = 0;
```

```
      x = 0;
    end
  //
  //
  //
  always
    begin
      //
      // Start after a positive edge from REQ!.
      //
      @(posedge REQ);
      # 10;
      //
      // Define the sector file name.
      //
      x = SECTOR;
      sector_0 = x%10;
      x = x/10;
      sector_1 = x%10;
      x = x/10;
      sector_2 = x%10;
      x = x/10;
      sector_3 = x%10;
      x = x/10;
      sector_4 = x%10;
      x = x/10;
      sector_5 = x%10;
      x = x/10;
      sector_6 = x%10;
      x = x/10;
      sector_7 = x%10;
      x = x/10;
      sector_8 = x%10;
      //
      // The string starts from the right to the left!
      //
      filename[12*8+7:12*8] = sector_8 + 8'd48;
      filename[11*8+7:11*8] = sector_7 + 8'd48;
      filename[10*8+7:10*8] = sector_6 + 8'd48;
      filename[9*8+7:9*8] = sector_5 + 8'd48;
```

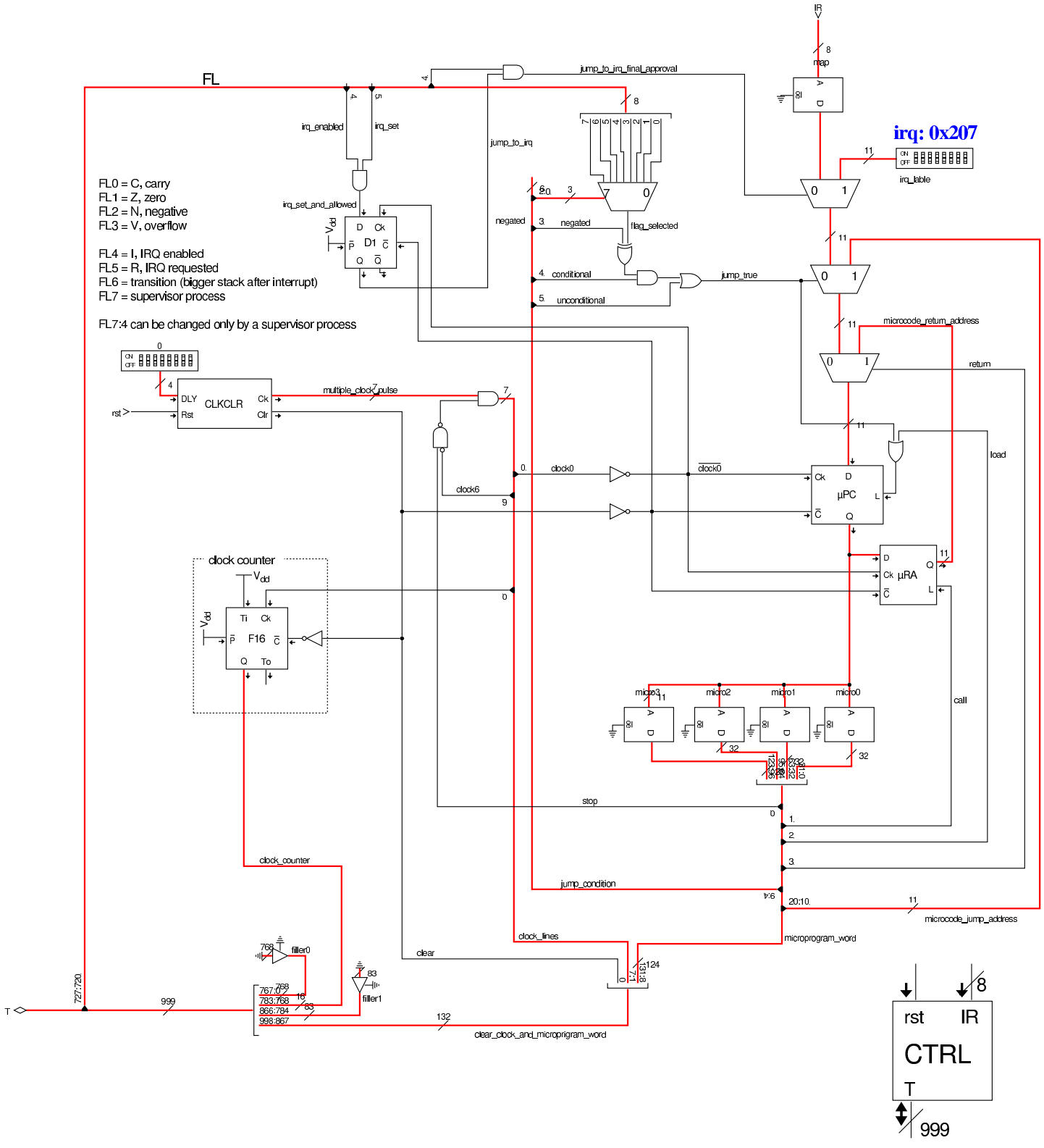
```
      filename[8*8+7:8*8] = sector_4 + 8'd48;
      filename[7*8+7:7*8] = sector_3 + 8'd48;
      filename[6*8+7:6*8] = sector_2 + 8'd48;
      filename[5*8+7:5*8] = sector_1 + 8'd48;
      filename[4*8+7:4*8] = sector_0 + 8'd48;
      //
      filename[21*8+7:21*8] = DRIVE + 8'd48;
      //
      if (WRITE)
        begin
          //
          // Put data inside the buffer.
          //
          buffer[BYTE] = DATA_IN;
          //
          // Save the buffer to disk.
          // Please remember that $writememh() must
          // be enabled inside Tkgate configuration!
          //
          $writememh(filename, buffer);
          //
          // Return the same data read.
          //
          _data_out = buffer[BYTE];
        end
      else
        begin
          //
          // Get data from disk to the buffer.
          //
          $readmemh(filename, buffer);
          //
          // Return the data required.
          //
          _data_out = buffer[BYTE];
        end
      //
      // Acknowledge.
      //
      _ack = 1;
```

```
      //
      // Wait the end of request (the negative edge)
      // before restarting the loop.
      //
      @(negedge REQ);
      # 10;
      //
      // Now become ready again.
      //
      _ack = 0;
    end
    //
  assign DATA_OUT = _data_out;
  assign ACK = _ack;
  //
endmodule
```

13.18 Module "CTRL"

The module **CTRL** is more complex than the previous version of this experimental project; apart the clock generator that is described in a previous section, there is also a new register, μ***RA***, that can be used to save a return address inside the microcode (the microcode might contain calls and returns, provided that they are not nested). However, the current microcode implementation does not use internal calls.

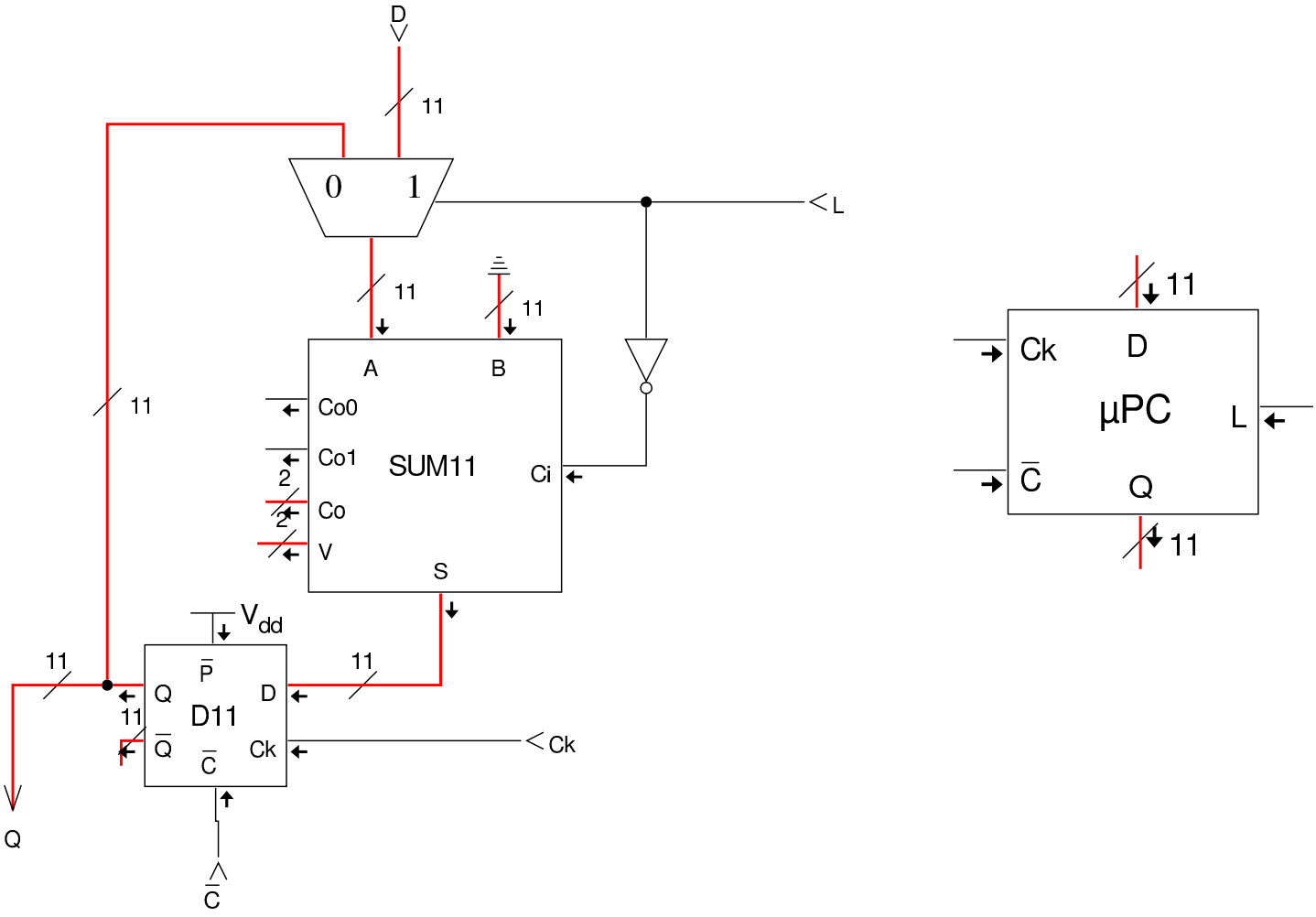
Figure 13.52. Module **CTRL**.



The microprogram counter (*µPC*) is of a 11-bit D flip-flop with an adder that increments the counter value when no load is requested. The adder that appears in the following figure is just a reduced module of the **SUM32**: there are many unused output

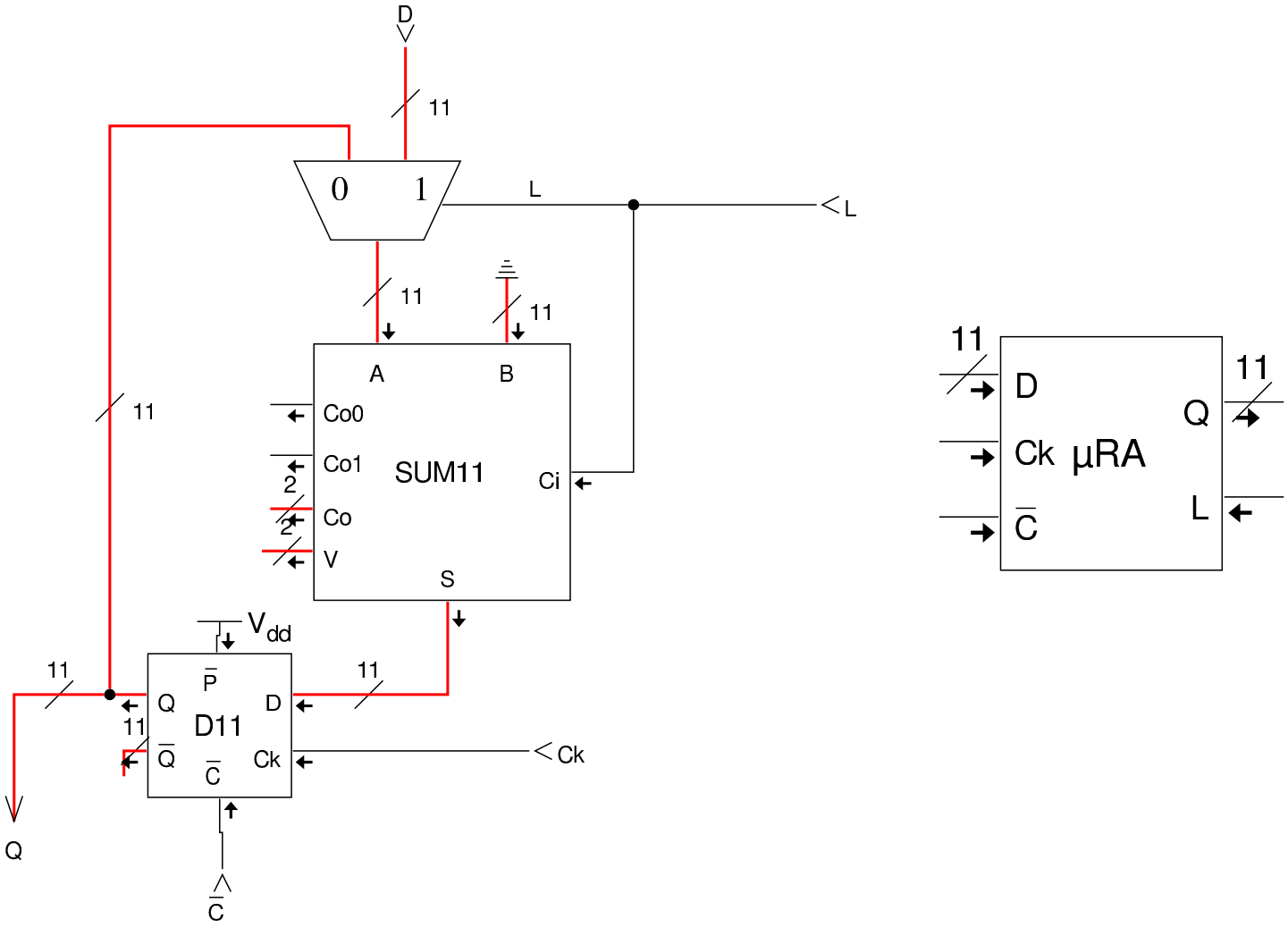
and the ***B*** input is also zeroed, because it is used only to increment by one, through the carry-in.

Figure 13.53. Module μ**PC**.



The register μ***RA*** (*microcode address*) is similar to the microprogram counter, but this time the loaded value is stored incremented, because when the current microcode address is loaded, the return address to be saved should be the next address.

Figure 13.54. Module μ**RA**.



13.19 Memory and microcode fields

The memory for this version of the project are described by the following lines of TKGate code. In this version of the project, every component has only the required functions and so the overall control lines are less then before.

Listing 13.55. Memory banks description for TKGate.

```
map        bank[9:0]      ctrl.map;
microcode  bank[31:0]     ctrl.micro0;
microcode  bank[63:32]    ctrl.micro1;
microcode  bank[95:64]    ctrl.micro2;
microcode  bank[123:96]   ctrl.micro3;
macrocode  bank[8:0]      ram.ram0;
```

Listing 13.56. Fields of the microcode word, for TKGate.

```
field CTRL[3:0]     = {nop=0x0, stop=0x1, call=0x2, load=0x4,
                       return=0x8};
field JUMP[9:4]     = {carry_t=0x00, zero_t=0x01, negative_t=0x02,
                       overflow_t=0x03, irq_enabled_t=0x04,
                       irq_requested_t=0x05, transition_t=0x06,
                       supervisor_t=0x07, carry_f=0x08, zero_f=0x09,
                       negative_f=0x0A, overflow_f=0x0B,
                       irq_enabled_f=0x0C, irq_requested_f=0x0D,
                       transition_f=0x0E, supervisor_f=0x0F,
                       conditional=0x10, unconditional=0x20};
field ADDR[20:10]   = {start=0x0};
field ALU[31:21]    = {rank8=0x000, rank16=0x001, rank24=0x002,
                       rank32=0x003, unsigned=0x000, signed=0x004,
                       set0=0x000, set1=0x008, equal=0x040, and=0x048,
                       or=0x050, xor=0x058, nxor=0x060, nor=0x068,
                       nand=0x070, not=0x078, add=0x080, sub=0x088,
                       addc=0x090, subb=0x098, lshl=0x0C0, lshcl=0x0C8,
                       lshr=0x100, lshcr=0x108, ashl=0x0C0, ashr=0x140,
                       rotl=0x180, rotr=0x1C0, carry0=0x200,
                       carry1=0x240, zero0=0x208, zero1=248,
                       negative0=0x210, negative1=0x250,
                       overflow0=0x218, overflow1=0x258, irqen0=0x220,
                       irqen1=0x260, irqreq0=0x228, irqreq1=0x268,
                       transition0=0x230, transition1=0x270,
                       supervisor0=0x238, supervisor1=0x278,
                       enable=0x400};
field Ar[33:32]     = {load=0x1, enable=0x2};
field Br[35:34]     = {load=0x1, enable=0x2};
field Cr[37:36]     = {load=0x1, enable=0x2};
field Dr[39:38]     = {load=0x1, enable=0x2};
field Ir[41:40]     = {load=0x1, enable=0x2};
field Ii[45:42]     = {zero=0x0};
field Jr[47:46]     = {load=0x1, enable=0x2};
field Ji[51:48]     = {zero=0x0};
field SPr[53:52]    = {load=0x1, enable=0x2};
field SPi[57:54]    = {zero=0x0};
field FPr[59:58]    = {load=0x1, enable=0x2};
field DA[62:60]     = {enable=0x1, i=0x0, j=0x2, sp=0x4, fp=0x6};
field DS0[64:63]    = {load=0x1, enable=0x2};
```

```
field DS1[66:65]    = {load=0x1, enable=0x2};
field CS0[68:67]    = {load=0x1, enable=0x2};
field CS1[70:69]    = {load=0x1, enable=0x2};
field PCr[72:71]    = {load=0x1, enable=0x2};
field PCi[76:73]    = {zero=0x0};
field SSP[78:77]    = {load=0x1, enable=0x2};
field TMP0[80:79]   = {load=0x1, enable=0x2};
field TMP1[82:81]   = {load=0x1, enable=0x2};
field TMP2[84:83]   = {load=0x1, enable=0x2};
field TMP3[86:85]   = {load=0x1, enable=0x2};
field TMP4[88:87]   = {load=0x1, enable=0x2};
field TMP5[90:89]   = {load=0x1, enable=0x2};
field M[100:91]     = {rank8=0x000,  rank16=0x001, rank24=0x002,
                       rank32=0x003, byte0=0x000,  byte1=0x004,
                       byte2=0x008,  byte3=0x00C,
                       data=0x000,   code=0x010,   ram_oe=0x020,
                       ram_we=0x040, md_enable=0x080, unsigned=0x000,
                       signed=0x100, ir_load=0x200};
field IVT[103:101]  = {load=0x1, enable=0x2, irq=0x0, int=0x4};
field IRQ[106:104]  = {mask_load=0x1, mask_enable=0x2, irq_done=0x4};
field FL[109:107]   = {enable=0x1, load_bus=0x2,
                      update_bus=0x4, load_alu=0x6};
field IOA[111:110]  = {load=0x1, enable=0x2};
field IOC[114:112]  = {load=0x1, enable=0x2, req=0x4};
field CNSv[122:115] = {zero=0x0};
field CNS[123:123]  = {enable=0x1};
```

13.20 Opcodes

«

This version of the simple CPU has a different set of opcodes, because the overall structure is different from the previous version, but the size remains of a single byte. Please notice that all the 256 combinations are used and there is no 'operror'.

Table 13.57. Opcodes.

	byte 0								byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12
	b7	b6	b5	b4	b3	b2	b1	b0												
0x00 00000000	nop																			
0x00 0000....	store %, %				A|B	A|B|C|D|I|J|SP|FP													
0x10 0001....	load %, %				A|B	A|B|C|D|I|J|SP|FP													
0x20 00100...	load8 0|1, (%)			rank8		sign	I|J|SP|FP													
0x28 00101...	load16 0|1, (%)			rank16		sign	I|J|SP|FP													
0x30 00110...	load24 0|1, (%)			rank24		sign	I|J|SP|FP													
0x38 00111...	load32 0|1, (%)			rank32		sign	I|J|SP|FP													

	byte 0								byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12
	b7	b6	b5	b4	b3	b2	b1	b0												
0x40 0100000.	load8 0|1, n					rank8		sign	const											
0x42 0100001.	load16 0|1, n					rank16		sign	const16											
0x44 0100010.	load24 0|1, n					rank24		sign	const24											
0x46 0100011.	load32 0|1, n					rank32		sign	const32											

	byte 0								byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12
	b7	b6	b5	b4	b3	b2	b1	b0												
0x48 01001...	lea %, (%)					A|B		I|J|SP|FP											
0x50 0101....	swap %, %				A|B	A|B|C|D|I|J|SP|FP													
0x60 011000..	store (%)				rank8			I|J|SP|FP												
0x64 011001..	store (%)				rank16			I|J|SP|FP												
0x68 011010..	store (%)				rank24			I|J|SP|FP												
0x6C 011011..	store (%)				rank32			I|J|SP|FP												
0x70 01110...	push %					A|B|C|D|I|J|SP|FP														
0x78 01111...	pop %					A|B|C|D|I|J|SP|FP														

	byte 0								byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12
	b7	b6	b5	b4	b3	b2	b1	b0												
0x80 100000..	equal						rank													
0x84 100001..	not						rank													
0x88 100010..	and						rank													
0x8C 100011..	nand						rank													
0x90 100100..	or						rank													
0x94 100101..	nor						rank													
0x98 100110..	xor						rank													
0x9C 100111..	nxor						rank													

	byte 0								byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12
	b7	b6	b5	b4	b3	b2	b1	b0												
0xA0 101000..	add						rank													
0xA4 101001..	sub						rank													
0xA8 101010..	addc						rank													
0xAC 101011..	subb						rank													
0xB0 101100..	lshl						rank													
0xB4 101101..	lshr						rank													
0xB8 101110..	ashl						rank													
0xBC 101111..	ashr						rank													

	byte 0								byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12
	b7	b6	b5	b4	b3	b2	b1	b0												
0xC0 110000..	rotcl						rank													
0xC4 110001..	rotcr						rank													
0xC8 110010..	rotl						rank													
0xCC 110011..	rotr						rank													
0xD0 110100..	utrunc						rank													
0xD4 110101..	strunc						rank													

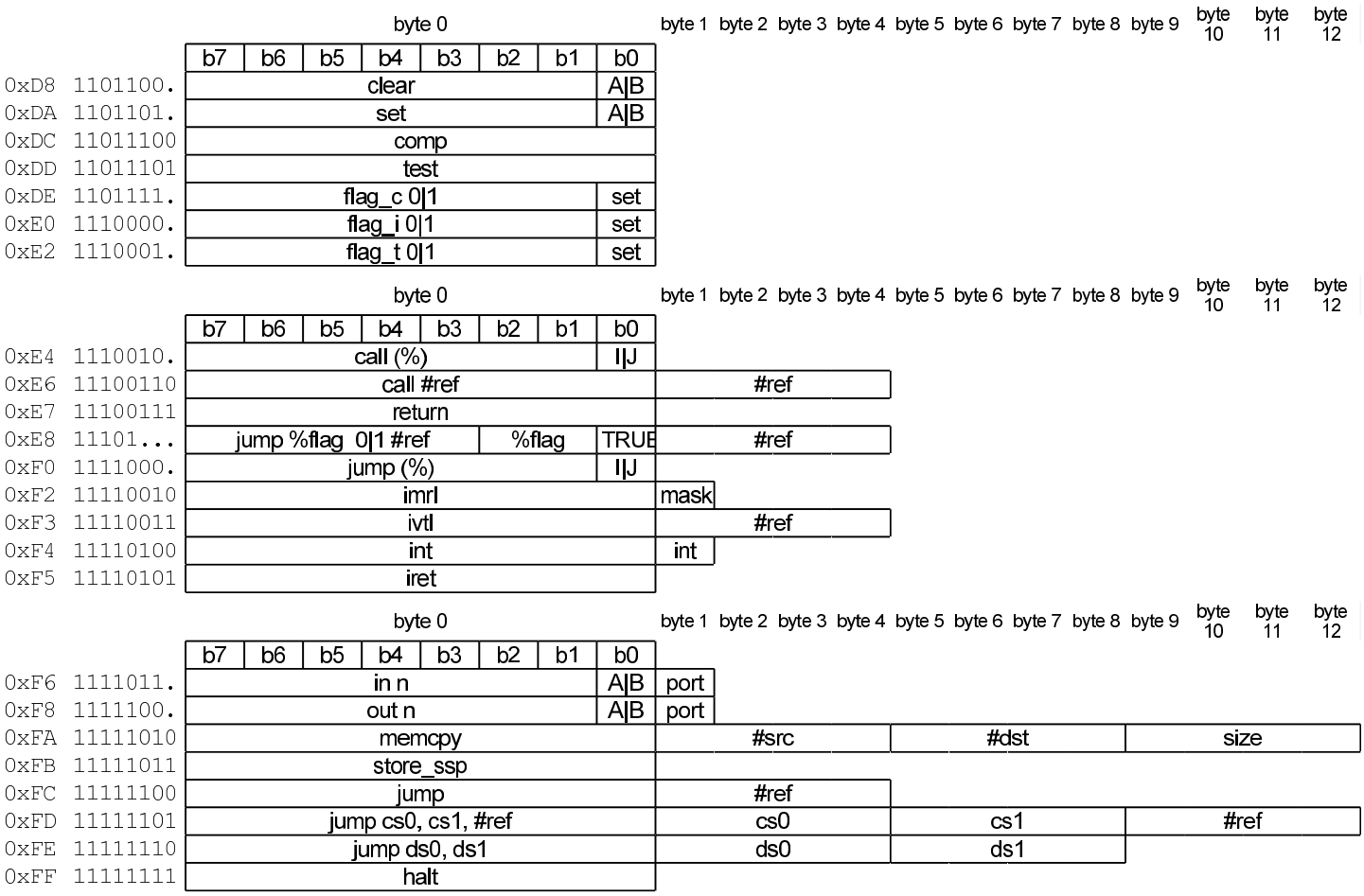


Table 13.58. Macrocode syntax.

Macrocode syntax	Description
`nop`	Not operate.

`store %A|%B, %A|%B|%C|%D|%I|%J|%SP|%FP`
`store %`***src***`, %`***dst***
`load %A|%B|%C|%D|%I|%J|%SP|%FP, %A|%B`
`load %`***dst***`, %`***src***
	Copy the ***src*** register content inside the ***dst*** register.

Macrocode syntax	Description
`load8 0, (%I|%J|%SP|%FP)`	Load an unsigned 8-bit value from the memory, at the address specified by the register that appears inside parenthesis plus ***J***, but if the register ***J*** is specified, no addition is made to the address. The value loaded from the memory is stored inside the register ***A***.
`load8 1, (%I|%J|%SP|%FP)`	Load a signed 8-bit value from the memory, at the address specified by the register that appears inside parenthesis plus ***J***, but if the register ***J*** is specified, no addition is made to the address. The value loaded from the memory is stored inside the register ***A***.

`load16 0|1, (%I|%J|%SP|%FP)`
`load24 0|1, (%I|%J|%SP|%FP)`
`load32 0|1, (%I|%J|%SP|%FP)`
	As above, but loading a 16, 24 or 32-bit value.

`load8 0|1,` ***value***	Load the specified value as unsigned or signed 8-bit, into the ***A*** register.
`load16 0|1,` ***value*** `load24 0|1,` ***value*** `load32, 0|1` ***value***	As above, but loading a 16, 24 or 32-bit value.

Macrocode syntax	Description
`lea %A|%B, (%I|%J|%SP|%FP)`	Load inside the register ***A*** or ***B*** the address contained inside the register in parenthesis, plus the value of ***J***. But if ***J*** was specified, it is just loaded as it is.
`swap %A|%B, (%A|%B|%C|%D|%I|%J|%SP|%FP)`	Swap the value of the first argument, with the one of the second.

`store8 (%I|%J|%SP|%FP)`
`store16 (%I|%J|%SP|%FP)`
`store24 (%I|%J|%SP|%FP)`
`store32 (%I|%J|%SP|%FP)`
	Store an 8, 16, 24 or 32-bit value from the register ***A***, to the address specified by the register in parenthesis. If the register is not ***J***, its value is added to ***J*** to find the destination address.

`push %A|%B, %A|%B|%C|%D|%I|%J|%SP|%FP` `pop %A|%B, %A|%B|%C|%D|%I|%J|%SP|%FP`	Push or pop the specified register. Push means that the ***SP*** register is decremented by 4 and at the memory location that ***SP*** points, the specified register is saved. Pop means to read from the address reported by the ***SP*** register a value that is then stored inside the register specified.

`equal8`
`equal16`
`equal24`
`equal32`
	A = ***A***, updating the flags. The number specify the rank; the result is sign-extended if necessary.

Macrocode syntax	Description

`not8`
`not16`
`not24`
`not32`
	A = ~***A***, updating the flags. The number specify the rank; the result is sign-extended if necessary.

`and8`
`and16`
`and24`
`and32`
	A = ***A*** & ***B***, updating the flags. The number specify the rank; the result is sign-extended if necessary.

`nand8`
`nand16`
`nand24`
`nand32`
	A = ~(***A*** & ***B***), updating the flags. The number specify the rank; the result is sign-extended if necessary.

`or8`
`or16`
`or24`
`or32`
	A = ***A*** | ***B***, updating the flags. The number specify the rank; the result is sign-extended if necessary.

Macrocode syntax	Description

`nor8`
`nor16`
`nor24`
`nor32`
	A = ~(***A*** | ***B***), updating the flags. The number specify the rank; the result is sign-extended if necessary.

`xor8`
`xor16`
`xor24`
`xor32`
	A = ***A*** ^ ***B***, updating the flags. The number specify the rank; the result is sign-extended if necessary.

`nxor8`
`nxor16`
`nxor24`
`nxor32`
	A = ~(***A*** ^ ***B***), updating the flags. The number specify the rank; the result is sign-extended if necessary.

`add8`
`add16`
`add24`
`add32`
	A = ***A*** + ***B***, updating the flags. The number specify the rank; the result is sign-extended if necessary.

Macrocode syntax	Description

`sub8`
`sub16`
`sub24`
`sub32`
	A **=** ***A*** **+** ***B***, updating the flags. The number specify the rank; the result is sign-extended if necessary.

`addc8`
`addc16`
`addc24`
`addc32`
	A **=** ***A*** **+** ***B*** **+** ***carry***, updating the flags. The number specify the rank; the result is sign-extended if necessary.

`subb8`
`subb16`
`subb24`
`subb32`
	A **=** ***A*** **−** ***B*** **−** ***borrow***, updating the flags. The number specify the rank; the result is sign-extended if necessary.

`lshl8`
`lshl16`
`lshl24`
`lshl32`
	A = lshl ***A*** (logic shift left), updating the flags. The number specify the rank; the result is sign-extended if necessary.

Macrocode syntax	Description

`lshr8`
`lshr16`
`lshr24`
`lshr32`
	A = lshr ***A*** (logic shift right), updating the flags. The number specify the rank; the result is sign-extended if necessary.

`ashl8`
`ashl16`
`ashl24`
`ashl32`
	A = ashl ***A*** (arithmetic shift left), updating the flags. The number specify the rank; the result is sign-extended if necessary.

`ashr8`
`ashr16`
`ashr24`
`ashr32`
	A = ashr ***A*** (arithmetic shift right), updating the flags. The number specify the rank; the result is sign-extended if necessary.

`rotcl8`
`rotcl16`
`rotcl24`
`rotcl32`
	A = rotcl ***A*** (rotate left with carry), updating the flags. The number specify the rank; the result is sign-extended if necessary.

Macrocode syntax	Description

`rotcr8`
`rotcr16`
`rotcr24`
`rotcr32`
	A = rotcr ***A*** (rotate right with carry), updating the flags. The number specify the rank; the result is sign-extended if necessary.

`rotl8`
`rotl16`
`rotl24`
`rotl32`
	A = rotl ***A*** (rotate left), updating the flags. The number specify the rank; the result is sign-extended if necessary.

`rotr8`
`rotr16`
`rotr24`
`rotr32`
	A = rotr ***A*** (rotate right), updating the flags. The number specify the rank; the result is sign-extended if necessary.

`utrunc8`
`utrunc16`
`utrunc24`
`utrunc32`
	Truncate the ***A*** value at the rank specified by the number attached to the name. The value is intended to be unsigned.

Macrocode syntax	Description

`strunc8`
`utrunc16`
`utrunc24`
`utrunc32`
	Truncate the ***A*** value at the rank specified by the number attached to the name. The value is intended to be signed.

`clear %A|%B` `set %A|%B`	Set the specified register to zero or to −1.
`comp`	Update the flags with the result of ***A*** − ***B***.
`test`	Update the flags with the result of ***A*** & ***B***.
`flag_c 0|1`	Clear or set the carry flag.
`flag_i 0|1`	Clear or set the interrupt allowed flag: can be used only by a supervisor process.
`flag_t 0|1`	Clear or set the transition flag: can be used only by a supervisor process.
`call` ***address*** `call #`***label*** `call (%I|%J)`	Call a procedure, specified by its numerical address, label or by registers ***I*** and ***J***.
`return`	Returns from a procedure call: it pops the ***PC*** value from the stack.

Macrocode syntax	Description
`jump` ***address*** `call #`***label*** `jump (%I|%J)`	Jump to the specified destination, unconditionally.
`jump 0|1, %carry|%borrow|%zero|` `%negative|%overflow,` ***address*** `|#`***label***	Jump to the specified destination, if the named flag is clear or set.
`imrl` ***n8***	Load the interrupt mask.
`ivtl` ***address*** `|#`***label***	Load the IVT: interrupt vector table. The argument is the interrupt vector table start address in memory.
`int` ***n8***	Software interrupt call: some registers are pushed onto the stack, depending on the current process privileges.
`iret`	Return from an interrupt, restoring the registers previously pushed by the **`int`** instruction or by another interruption. The instruction **`iret`**, if the stack is properly prepared, allows to a supervisor process to create a user process into particular code and data segments.
`jump` ***address0, address1***	This is an alternative way to change the data segment, assigning the first argument to register ***DS0*** and the second to ***DS1***. This instruction is available only to a supervisor process.

Macrocode syntax	Description
`jump` ***address0*** `,` ***address1*** `,` ***start*** `|` `#`***label***	This is an alternative way to change the code segment, assigning the first argument to register ***CS0***, the second to ***CS1*** and the third to ***PC***. This instruction is available only to a supervisor process.
`store_ssp`	Assign to register ***SSP*** the value of ***A***. The register ***SSP*** is used to save the supervisor stack pointer.
`memcpy` ***src*** `,` ***dst*** `,` ***count***	Copy ***count*** bytes from ***src*** to ***dst***. Source and destination can be specified numerically or by labels.
`in %A|%B,` ***n***	Read, from the I/O address specified, a byte that is copied inside register ***A*** or ***B***. Also the flags are updated.
`out %A|%B,` ***n***	Write, to the I/O address specified, a byte that is read from register ***A*** or ***B***.
`halt`	Stop the system.

Listing 13.59. Registers, flags and opcodes encoded for TKGate.

```
registers A=0, B=1, C=2, D=3, I=4, J=5, SP=6, FP=7;

registers carry=0, borrow=0, zero=1, negative=2, overflow=3;
//-----------------------------------------------------------------
op nop {
  map nop:         0x00;  // not operate
  operands { - = { +0=0x00; }; };
};
op store {
  map nop:         0x00;  // 00000000 = store %A,%A  = nop
  map st_a_b:      0x01;  // 00000001 = store %A,%B
  map st_a_c:      0x02;  // 00000010 = store %A,%C
```

```
  map st_a_d:      0x03;  // 00000011 = store %A,%D
  map st_a_i:      0x04;  // 00000100 = store %A,%I
  map st_a_j:      0x05;  // 00000101 = store %A,%J
  map st_a_sp:     0x06;  // 00000110 = store %A,%SP
  map st_a_fp:     0x07;  // 00000111 = store %A,%FP
  map st_b_a:      0x08;  // 00001000 = store %B,%A
  map nop:         0x09;  // 00001001 = store %B,%B  = nop
  map st_b_c:      0x0A;  // 00001010 = store %B,%C
  map st_b_d:      0x0B;  // 00001011 = store %B,%D
  map st_b_i:      0x0C;  // 00001100 = store %B,%I
  map st_b_j:      0x0D;  // 00001101 = store %B,%J
  map st_b_sp:     0x0E;  // 00001110 = store %B,%SP
  map st_b_fp:     0x0F;  // 00001111 = store %B,%FP
  operands {
    %1,%2   = { +0=0x00; +0[3:3]=%1; +0[2:0]=%2; };
  };
};
op load {
  map nop:         0x10;  // 00010000 = load %A,%A  = nop
  map ld_a_b:      0x11;  // 00010001 = load %A,%B
  map ld_a_c:      0x12;  // 00010010 = load %A,%C
  map ld_a_d:      0x13;  // 00010011 = load %A,%D
  map ld_a_i:      0x14;  // 00010100 = load %A,%I
  map ld_a_j:      0x15;  // 00010101 = load %A,%J
  map ld_a_sp:     0x16;  // 00010110 = load %A,%SP
  map ld_a_fp:     0x17;  // 00010111 = load %A,%FP
  map ld_b_a:      0x18;  // 00011000 = load %B,%A
  map nop:         0x19;  // 00011001 = load %B,%B  = nop
  map ld_b_c:      0x1A;  // 00011010 = load %B,%C
  map ld_b_d:      0x1B;  // 00011011 = load %B,%D
  map ld_b_i:      0x1C;  // 00011100 = load %B,%I
  map ld_b_j:      0x1D;  // 00011101 = load %B,%J
  map ld_b_sp:     0x1E;  // 00011110 = load %B,%SP
  map ld_b_fp:     0x1F;  // 00011111 = load %B,%FP
  operands {
    %1,%2   = { +0=0x10; +0[3:3]=%1; +0[2:0]=%2; };
  };
};
//
op load8 {
```

```
  map ld8ur_i:     0x20;  // 00100000 = load8 0, %I  = A <- (I+J)  unsign
  map ld8ur_j:     0x21;  // 00100001 = load8 0, %J  = A <- (J)    unsign
  map ld8ur_sp:    0x22;  // 00100010 = load8 0, %SP = A <- (SP+J) unsign
  map ld8ur_fp:    0x23;  // 00100011 = load8 0, %FP = A <- (FP+J) unsign
  map ld8sr_i:     0x24;  // 00100100 = load8 1, %I  = A <- (I+J)  signed
  map ld8sr_j:     0x25;  // 00100101 = load8 1, %J  = A <- (J)    signed
  map ld8sr_sp:    0x26;  // 00100110 = load8 1, %SP = A <- (SP+J) signed
  map ld8sr_fp:    0x27;  // 00100111 = load8 1, %FP = A <- (FP+J) signed
  map ld8u:        0x40;  // 01000000 = load8 0, n   = A <- n      unsign
  map ld8s:        0x41;  // 01000001 = load8 1, n   = A <- n      signed
  operands {
    #1,(%2) = { +0=0x20; +0[2:2]=#1; +0[1:0]=%2; };
    #1,#2   = { +0=0x40; +0[0:0]=#1; +1=#2[7:0]; };
  };
};
op load16 {
  map ld16ur_i:    0x28;  // 00101000 = load16 0, %I  = A <- (I+J)  unsign
  map ld16ur_j:    0x29;  // 00101001 = load16 0, %J  = A <- (J)    unsign
  map ld16ur_sp:   0x2A;  // 00101010 = load16 0, %SP = A <- (SP+J) unsign
  map ld16ur_fp:   0x2B;  // 00101011 = load16 0, %FP = A <- (FP+J) unsign
  map ld16sr_i:    0x2C;  // 00101100 = load16 1, %I  = A <- (I+J)  signed
  map ld16sr_j:    0x2D;  // 00101101 = load16 1, %J  = A <- (I+J)  signed
  map ld16sr_sp:   0x2E;  // 00101110 = load16 1, %SP = A <- (SP+J) signed
  map ld16sr_fp:   0x2F;  // 00101111 = load16 1, %FP = A <- (FP+J) signed
  map ld16u:       0x42;  // 01000010 = load16 0, n   = A <- n      unsign
  map ld16s:       0x43;  // 01000011 = load16 1, n   = A <- n      signed
  operands {
    #1,(%2) = { +0=0x28; +0[2:2]=#1; +0[1:0]=%2; };
    #1,#2   = { +0=0x42; +0[0:0]=#1; +1=#2[15:8]; +2=#2[7:0]; };
  };
};
op load24 {
  map ld24ur_i:    0x30;  // 00110000 = load24 0, %I  = A <- (%+J)  unsign
  map ld24ur_j:    0x31;  // 00110001 = load24 0, %J  = A <- (J)    unsign
  map ld24ur_sp:   0x32;  // 00110010 = load24 0, %SP = A <- (SP+J) unsign
  map ld24ur_fp:   0x33;  // 00110011 = load24 0, %FP = A <- (FP+J) unsign
  map ld24sr_i:    0x34;  // 00110100 = load24 1, %I  = A <- (I+J)  signed
  map ld24sr_j:    0x35;  // 00110101 = load24 1, %J  = A <- (J)    signed
  map ld24sr_sp:   0x36;  // 00110110 = load24 1, %SP = A <- (SP+J) signed
  map ld24sr_fp:   0x37;  // 00110111 = load24 1, %FP = A <- (FP+J) signed
```

```
  map ld24u:      0x44;  // 01000100 = load24 0, n   = A <- n      unsign
  map ld24s:      0x45;  // 01000101 = load24 1, n   = A <- n      signed
  operands {
    #1,(%2) = { +0=0x30; +0[2:2]=#1; +0[1:0]=%2; };
    #1,#2   = { +0=0x44; +0[0:0]=#1; +1=#2[24:16]; +2=#2[15:8];
                +3=#2[7:0]; };
  };
};
op load32 {
  map ld32ur_i:   0x38;  // 00111000 = load32 0, %I  = A <- (I+J)  unsign
  map ld32ur_j:   0x39;  // 00111001 = load32 0, %J  = A <- (J)    unsign
  map ld32ur_sp:  0x3A;  // 00111010 = load32 0, %SP = A <- (SP+J) unsign
  map ld32ur_fp:  0x3B;  // 00111011 = load32 0, %FP = A <- (FP+J) unsign
  map ld32sr_i:   0x3C;  // 00111100 = load32 1, %I  = A <- (I+J)  signed
  map ld32sr_j:   0x3D;  // 00111101 = load32 1, %J  = A <- (J)    signed
  map ld32sr_sp:  0x3E;  // 00111110 = load32 1, %SP = A <- (SP+J) signed
  map ld32sr_fp:  0x3F;  // 00111111 = load32 1, %FP = A <- (FP+J) signed
  map ld32u:      0x46;  // 01000110 = load32 0, n   = A <- n      unsign
  map ld32s:      0x47;  // 01000111 = load32 1, n   = A <- n      signed
  operands {
    #1,(%2) = { +0=0x38; +0[2:2]=#1; +0[1:0]=%2; };
    #1,#2   = { +0=0x46; +0[0:0]=#1;
                +1=#2[31:24]; +2=#2[23:16]; +3=#2[15:8]; +4=#2[7:0]; };
  };
};
op lea {
  map lea_a_i:    0x48;  // 01001000 = lea %A, (%I)  = A <- lea (I+J)
  map lea_a_j:    0x49;  // 01001001 = lea %A, (%J)  = A <- lea (J)
  map lea_a_sp:   0x4A;  // 01001010 = lea %A, (%SP) = A <- lea (SP+J)
  map lea_a_fp:   0x4B;  // 01001011 = lea %A, (%FP) = A <- lea (FP+J)
  map lea_b_i:    0x4C;  // 01001100 = lea %B, (%I)  = B <- lea (I+J)
  map lea_b_j:    0x4D;  // 01001101 = lea %B, (%J)  = B <- lea (J)
  map lea_b_sp:   0x4E;  // 01001110 = lea %B, (%SP) = B <- lea (SP+J)
  map lea_b_fp:   0x4F;  // 01001111 = lea %B, (%FP) = B <- lea (FP+J)
  operands {
    %1,(%2) = { +0=0x48; +0[2:2]=%1; +0[1:0]=%2; };
  };
};
op swap {
  map swap_a_a:   0x50;  // 01010000 = swap %A, %A      not implemented
```

```
  map swap_a_b:    0x51;  // 01010001 = swap %A, %B      not implemented
  map swap_a_c:    0x52;  // 01010010 = swap %A, %C      not implemented
  map swap_a_d:    0x53;  // 01010011 = swap %A, %D      not implemented
  map swap_a_i:    0x54;  // 01010100 = swap %A, %I      not implemented
  map swap_a_j:    0x55;  // 01010101 = swap %A, %J      not implemented
  map swap_a_sp:   0x56;  // 01010110 = swap %A, %SP     not implemented
  map swap_a_fp:   0x57;  // 01010111 = swap %A, %FP     not implemented
  map swap_b_a:    0x58;  // 01011000 = swap %B, %A      not implemented
  map swap_b_b:    0x59;  // 01011001 = swap %B, %B      not implemented
  map swap_b_c:    0x5A;  // 01011010 = swap %B, %C      not implemented
  map swap_b_d:    0x5B;  // 01011011 = swap %B, %D      not implemented
  map swap_b_i:    0x5C;  // 01011100 = swap %B, %I      not implemented
  map swap_b_j:    0x5D;  // 01011101 = swap %B, %J      not implemented
  map swap_b_sp:   0x5E;  // 01011110 = swap %B, %SP     not implemented
  map swap_b_fp:   0x5F;  // 01011111 = swap %B, %FP     not implemented
  operands {
    %1,%2   = { +0=0x50; +0[3:3]=%1; +0[2:0]=%2; };
  };
};
op store8 {
  map st8r_i:      0x60;  // 01100000 = store8 (%I)    = M(I+J) <- A
  map st8r_j:      0x61;  // 01100001 = store8 (%J)    = M(J) <- A
  map st8r_sp:     0x62;  // 01100010 = store8 (%SP)   = M(SP+J) <- A
  map st8r_fp:     0x63;  // 01100011 = store8 (%FP)   = M(FP+J) <- A
  operands {
    (%1) = { +0=0x60; +0[1:0]=%1; };
  };
};
op store16 {
  map st16r_i:     0x64;  // 01100100 = store16 (%I)   = M(I+J) <- A
  map st16r_j:     0x65;  // 01100101 = store16 (%J)   = M(J) <- A
  map st16r_sp:    0x66;  // 01100110 = store16 (%SP)  = M(SP+J) <- A
  map st16r_fp:    0x67;  // 01100111 = store16 (%FP)  = M(FP+J) <- A
  operands {
    (%1) = { +0=0x64; +0[1:0]=%1; };
  };
};
op store24 {
  map st24r_i:     0x68;  // 01101000 = store24 (%I)   = M(I+J) <- A
  map st24r_j:     0x69;  // 01101001 = store24 (%J)   = M(J) <- A
```

```
  map st24r_sp:    0x6A;  // 01101010 = store24 (%SP) = M(SP+J) <- A
  map st24r_fp:    0x6B;  // 01101011 = store24 (%FP) = M(FP+J) <- A
  operands {
    (%1) = { +0=0x68; +0[1:0]=%1; };
  };
};
op store32 {
  map st32r_i:     0x6C;  // 01101100 = store32 (%I)  = M(I+J) <- A
  map st32r_j:     0x6D;  // 01101101 = store32 (%J)  = M(J) <- A
  map st32r_sp:    0x6E;  // 01101110 = store32 (%SP) = M(SP+J) <- A
  map st32r_fp:    0x6F;  // 01101111 = store32 (%FP) = M(FP+J) <- A
  operands {
    (%1) = { +0=0x6C; +0[1:0]=%1; };
  };
};
op push {
  map push_a:      0x70;  // 01110000 = push %A
  map push_b:      0x71;  // 01110001 = push %B
  map push_c:      0x72;  // 01110010 = push %C
  map push_d:      0x73;  // 01110011 = push %D
  map push_i:      0x74;  // 01110100 = push %I
  map push_j:      0x75;  // 01110101 = push %J
  map push_sp:     0x76;  // 01110110 = push %SP
  map push_fp:     0x77;  // 01110111 = push %FP
  operands {
    %1 = { +0=0x70; +0[2:0]=%1; };
  };
};
op pop {
  map pop_a:       0x78;  // 01111000 = pop %A
  map pop_b:       0x79;  // 01111001 = pop %B
  map pop_c:       0x7A;  // 01111010 = pop %C
  map pop_d:       0x7B;  // 01111011 = pop %D
  map pop_i:       0x7C;  // 01111100 = pop %I
  map pop_j:       0x7D;  // 01111101 = pop %J
  map pop_sp:      0x7E;  // 01111110 = pop %SP
  map pop_fp:      0x7F;  // 01111111 = pop %FP
  operands {
    %1 = { +0=0x78; +0[2:0]=%1; };
  };
```

```
};
op equal8 {
  map equal8s:     0x80;  // 10000000 = equal8  %A = A <- A
  operands { - = { +0=0x80; }; };
};
op equal16 {
  map equal16s:    0x81;  // 10000001 = equal16 %A = A <- A
  operands { - = { +0=0x81; }; };
};
op equal24 {
  map equal24s:    0x82;  // 10000010 = equal24 %A = A <- A
  operands { - = { +0=0x82; }; };
};
op equal32 {
  map equal32s:    0x83;  // 10000011 = equal32 %A = A <- A
  operands { - = { +0=0x83; }; };
};
op not8 {
  map not8s:       0x84;  // 10000100 = not8  %A = A <- ~A
  operands { - = { +0=0x84; }; };
};
op not16 {
  map not16s:      0x85;  // 10000101 = not16 %A = A <- ~A
  operands { - = { +0=0x85; }; };
};
op not24 {
  map not24s:      0x86;  // 10000110 = not24 %A = A <- ~A
  operands { - = { +0=0x86; }; };
};
op not32 {
  map not32s:      0x87;  // 10000111 = not32 %A = A <- ~A
  operands { - = { +0=0x87; }; };
};
op and8 {
  map and8s:       0x88;  // 10001000 = and8  %A = A <- A&B
  operands { - = { +0=0x88; }; };
};
op and16 {
  map and16s:      0x89;  // 10001001 = and16 %A = A <- A&B
  operands { - = { +0=0x89; }; };
```

```
};
op and24 {
  map and24s:      0x8A;  // 10001010 = and24 %A = A <- A&B
  operands { - = { +0=0x8A; }; };
};
op and32 {
  map and32s:      0x8B;  // 10001011 = and32 %A = A <- A&B
  operands { - = { +0=0x8B; }; };
};
op nand8 {
  map nand8s:      0x8C;  // 10001100 = nand8  %A = A <- ~A&B
  operands { - = { +0=0x8C; }; };
};
op nand16 {
  map nand16s:     0x8D;  // 10001101 = nand16 %A = A <- ~A&B
  operands { - = { +0=0x8D; }; };
};
op nand24 {
  map nand24s:     0x8E;  // 10001110 = nand24 %A = A <- ~A&B
  operands { - = { +0=0x8E; }; };
};
op nand32 {
  map nand32s:     0x8F;  // 10001111 = nand32 %A = A <- ~A&B
  operands { - = { +0=0x8F; }; };
};
op or8 {
  map or8s:        0x90;  // 10010000 = or8  %A = A <- A|B
  operands { - = { +0=0x90; }; };
};
op or16 {
  map or16s:       0x91;  // 10010001 = or16 %A = A <- A|B
  operands { - = { +0=0x91; }; };
};
op or24 {
  map or24s:       0x92;  // 10010010 = or24 %A = A <- A|B
  operands { - = { +0=0x92; }; };
};
op or32 {
  map or32s:       0x93;  // 10010011 = or32 %A = A <- A|B
  operands { - = { +0=0x93; }; };
```

```
};
op nor8 {
  map nor8s:       0x94;  // 10010100 = nor8  %A = A <- ~A|B
  operands { - = { +0=0x94; }; };
};
op nor16 {
  map nor16s:      0x95;  // 10010101 = nor16 %A = A <- ~A|B
  operands { - = { +0=0x95; }; };
};
op nor24 {
  map nor24s:      0x96;  // 10010110 = nor24 %A = A <- ~A|B
  operands { - = { +0=0x96; }; };
};
op nor32 {
  map nor32s:      0x97;  // 10010111 = nor32 %A = A <- ~A|B
  operands { - = { +0=0x97; }; };
};
op xor8 {
  map xor8s:       0x98;  // 10011000 = xor8  %A = A <- A^B
  operands { - = { +0=0x98; }; };
};
op xor16 {
  map xor16s:      0x99;  // 10011001 = xor16 %A = A <- A^B
  operands { - = { +0=0x99; }; };
};
op xor24 {
  map xor24s:      0x9A;  // 10011010 = xor24 %A = A <- A^B
  operands { - = { +0=0x9A; }; };
};
op xor32 {
  map xor32s:      0x9B;  // 10011011 = xor32 %A = A <- A^B
  operands { - = { +0=0x9B; }; };
};
op nxor8 {
  map nxor8s:      0x9C;  // 10011000 = nxor8  %A = A <- ~A^B
  operands { - = { +0=0x9C; }; };
};
op nxor16 {
  map nxor16s:     0x9D;  // 10011001 = nxor16 %A = A <- ~A^B
  operands { - = { +0=0x9D; }; };
```

```
};
op nxor24 {
  map nxor24s:     0x9E;  // 10011010 = nxor24 %A = A <- ~A^B
  operands { - = { +0=0x9E; }; };
};
op nxor32 {
  map nxor32s:     0x9F;  // 10011011 = nxor32 %A = A <- ~A^B
  operands { - = { +0=0x9F; }; };
};
op add8 {
  map add8s:       0xA0;  // 10100000 = add8  %A = A <- A+B
  operands { - = { +0=0xA0; }; };
};
op add16 {
  map add16s:      0xA1;  // 10100001 = add16 %A = A <- A+B
  operands { - = { +0=0xA1; }; };
};
op add24 {
  map add24s:      0xA2;  // 10100010 = add24 %A = A <- A+B
  operands { - = { +0=0xA2; }; };
};
op add32 {
  map add32s:      0xA3;  // 10100011 = add32 %A = A <- A+B
  operands { - = { +0=0xA3; }; };
};
op sub8 {
  map sub8s:       0xA4;  // 10100000 = sub8  %A = A <- A-B
  operands { - = { +0=0xA4; }; };
};
op sub16 {
  map sub16s:      0xA5;  // 10100001 = sub16 %A = A <- A-B
  operands { - = { +0=0xA5; }; };
};
op sub24 {
  map sub24s:      0xA6;  // 10100010 = sub24 %A = A <- A-B
  operands { - = { +0=0xA6; }; };
};
op sub32 {
  map sub32s:      0xA7;  // 10100011 = sub32 %A = A <- A-B
  operands { - = { +0=0xA7; }; };
```

```
};
op addc8 {
  map addc8s:      0xA8;  // 10101000 = addc8  %A = A <- A+B+c
  operands { - = { +0=0xA8; }; };
};
op addc16 {
  map addc16s:     0xA9;  // 10101001 = addc16 %A = A <- A+B+c
  operands { - = { +0=0xA9; }; };
};
op addc24 {
  map addc24s:     0xAA;  // 10101010 = addc24 %A = A <- A+B+c
  operands { - = { +0=0xAA; }; };
};
op addc32 {
  map addc32s:     0xAB;  // 10101011 = addc32 %A = A <- A+B+c
  operands { - = { +0=0xAB; }; };
};
op subb8 {
  map subb8s:      0xAC;  // 10101100 = subb8  %A = A <- A-B-b
  operands { - = { +0=0xAC; }; };
};
op subb16 {
  map subb16s:     0xAD;  // 10101101 = subb16 %A = A <- A-B-b
  operands { - = { +0=0xAD; }; };
};
op subb24 {
  map subb24s:     0xAE;  // 10101110 = subb24 %A = A <- A-B-b
  operands { - = { +0=0xAE; }; };
};
op subb32 {
  map subb32s:     0xAF;  // 10101111 = subb32 %A = A <- A-B-b
  operands { - = { +0=0xAF; }; };
};
op lshl8 {
  map lshl8s:      0xB0;  // 10110000 = lshl8  %A = A <- lshl A
  operands { - = { +0=0xB0; }; };
};
op lshl16 {
  map lshl16s:     0xB1;  // 10101101 = lshl16 %A = A <- lshl A
  operands { - = { +0=0xB1; }; };
```

```
};
op lshl24 {
  map lshl24s:    0xB2;  // 10101110 = lshl24 %A = A <- lshl A
  operands { - = { +0=0xB2; }; };
};
op lshl32 {
  map lshl32s:    0xB3;  // 10101111 = lshl32 %A = A <- lshl A
  operands { - = { +0=0xB3; }; };
};
op lshr8 {
  map lshr8s:     0xB4;  // 10110000 = lshr8  %A = A <- lshr A
  operands { - = { +0=0xB4; }; };
};
op lshr16 {
  map lshr16s:    0xB5;  // 10110001 = lshr16 %A = A <- lshr A
  operands { - = { +0=0xB5; }; };
};
op lshr24 {
  map lshr24s:    0xB6;  // 10110010 = lshr24 %A = A <- lshr A
  operands { - = { +0=0xB6; }; };
};
op lshr32 {
  map lshr32s:    0xB7;  // 10110011 = lshr32 %A = A <- lshr A
  operands { - = { +0=0xB7; }; };
};
op ashl8 {
  map ashl8s:     0xB8;  // 10111000 = ash18  %A = A <- ashl A
  operands { - = { +0=0xB8; }; };
};
op ashl16 {
  map ashl16s:    0xB9;  // 10111001 = ash116 %A = A <- ashl A
  operands { - = { +0=0xB9; }; };
};
op ashl24 {
  map ashl24s:    0xBA;  // 10111010 = ash124 %A = A <- ashl A
  operands { - = { +0=0xBA; }; };
};
op ashl32 {
  map ashl32s:    0xBB;  // 10111011 = ash132 %A = A <- ashl A
  operands { - = { +0=0xBB; }; };
```

```
};
op ashr8 {
  map ashr8s:      0xBC;  // 10111100 = ashr8  %A = A <- ashr A
  operands { - = { +0=0xBC; }; };
};
op ashr16 {
  map ashr16s:     0xBD;  // 10111101 = ashr16 %A = A <- ashr A
  operands { - = { +0=0xBD; }; };
};
op ashr24 {
  map ashr24s:     0xBE;  // 10111110 = ashr24 %A = A <- ashr A
  operands { - = { +0=0xBE; }; };
};
op ashr32 {
  map ashr32s:     0xBF;  // 10111111 = ashr32 %A = A <- ashr A
  operands { - = { +0=0xBF; }; };
};
op rotcl8 {
  map rotcl8s:     0xC0;  // 11000000 = rotcl8  %A = A <- rotcl A
  operands { - = { +0=0xC0; }; };
};
op rotcl16 {
  map rotcl16s:    0xC1;  // 11000001 = rotcl16 %A = A <- rotcl A
  operands { - = { +0=0xC1; }; };
};
op rotcl24 {
  map rotcl24s:    0xC2;  // 11000010 = rotcl24 %A = A <- rotcl A
  operands { - = { +0=0xC2; }; };
};
op rotcl32 {
  map rotcl32s:    0xC3;  // 11000011 = rotcl32 %A = A <- rotcl A
  operands { - = { +0=0xC3; }; };
};
op rotcr8 {
  map rotcr8s:     0xC4;  // 11000100 = rotcr8  %A = A <- rotcr A
  operands { - = { +0=0xC4; }; };
};
op rotcr16 {
  map rotcr16s:    0xC5;  // 11000101 = rotcr16 %A = A <- rotcr A
  operands { - = { +0=0xC5; }; };
```

```
};
op rotcr24 {
  map rotcr24s:    0xC6;  // 11000110 = rotcr24 %A = A <- rotcr A
  operands { - = { +0=0xC6; }; };
};
op rotcr32 {
  map rotcr32s:    0xC7;  // 11000111 = rotcr32 %A = A <- rotcr A
  operands { - = { +0=0xC7; }; };
};
op rotl8 {
  map rotl8s:      0xC8;  // 11001000 = rotl8  %A = A <- rotl A
  operands { - = { +0=0xC8; }; };
};
op rotl16 {
  map rotl16s:     0xC9;  // 11001001 = rotl16 %A = A <- rotl A
  operands { - = { +0=0xC9; }; };
};
op rotl24 {
  map rotl24s:     0xCA;  // 11001010 = rotl24 %A = A <- rotl A
  operands { - = { +0=0xCA; }; };
};
op rotl32 {
  map rotl32s:     0xCB;  // 11001011 = rotl32 %A = A <- rotl A
  operands { - = { +0=0xCB; }; };
};
op rotr8 {
  map rotr8s:      0xCC;  // 11001100 = rotr8  %A = A <- rotr A
  operands { - = { +0=0xCC; }; };
};
op rotr16 {
  map rotr16s:     0xCD;  // 11001101 = rotr16 %A = A <- rotr A
  operands { - = { +0=0xCD; }; };
};
op rotr24 {
  map rotr24s:     0xCE;  // 11001110 = rotr24 %A = A <- rotr A
  operands { - = { +0=0xCE; }; };
};
op rotr32 {
  map rotr32s:     0xCF;  // 11001111 = rotr32 %A = A <- rotr A
  operands { - = { +0=0xCF; }; };
```

```
};
op utrunc8 {
  map utrunc8:    0xD0;  // 11010000 = utrunc8  %A = A <- utrunc A
  operands { - = { +0=0xD0; }; };
};
op utrunc16 {
  map utrunc16:   0xD1;  // 11010001 = utrunc16 %A = A <- utrunc A
  operands { - = { +0=0xD1; }; };
};
op utrunc24 {
  map utrunc24:   0xD2;  // 11010010 = utrunc24 %A = A <- utrunc A
  operands { - = { +0=0xD2; }; };
};
op utrunc32 {
  map utrunc32:   0xD3;  // 11010011 = utrunc32 %A = A <- utrunc A
  operands { - = { +0=0xD3; }; };
};
op strunc8 {
  map strunc8:    0xD4;  // 11010100 = strunc8  %A = A <- strunc A
  operands { - = { +0=0xD4; }; };
};
op strunc16 {
  map strunc16:   0xD5;  // 11010101 = strunc16 %A = A <- strunc A
  operands { - = { +0=0xD5; }; };
};
op strunc24 {
  map strunc24:   0xD6;  // 11010110 = strunc24 %A = A <- strunc A
  operands { - = { +0=0xD6; }; };
};
op strunc32 {
  map strunc32:   0xD7;  // 11010111 = strunc32 %A = A <- strunc A
  operands { - = { +0=0xD7; }; };
};
op clear {
  map clear_a:    0xD8;  // 11011000 = clear %A = A <- 0
  map clear_b:    0xD9;  // 11011001 = clear %B = B <- 0
  operands {
    %1 = { +0=0xD8; +0[0:0]=%1; };
  };
};
```

```
op set {
  map set_a:       0xDA;  // 11011010 = set %A = A <- -1
  map set_b:       0xDB;  // 11011011 = set %B = B <- -1
  operands {
    %1 = { +0=0xDA; +0[0:0]=%1; };
  };
};
op comp {
  map comp:        0xDC;  // 11011100 = update FL with A-B
  operands { - = { +0=0xDC; }; };
};
op test {
  map test:        0xDD;  // 11011101 = update FL with A&B
  operands { - = { +0=0xDD; }; };
};
op flag_c {
  map flag_c_0:    0xDE;  // 11011110 = flag_c 0 = clear carry flag
  map flag_c_1:    0xDF;  // 11011111 = flag_c 1 = set carry flag
  operands {
    #1 = { +0=0xDE; +0[0:0]=#1; };
  };
};
op flag_i {
  map flag_i_0:    0xE0;  // 11100000 = flag_i 0 = clear irq enable flag
  map flag_i_1:    0xE1;  // 11100001 = flag_i 1 = set irq enable flag
  operands {
    #1 = { +0=0xE0; +0[0:0]=#1; };
  };
};
op flag_t {
  map flag_t_0:    0xE2;  // 11100010 = flag_t 0 = clear supervisor flag
  map flag_t_1:    0xE3;  // 11100011 = flag_t 1 = set supervisor flag
  operands {
    #1 = { +0=0xE2; +0[0:0]=#1; };
  };
};
op call {
  map callr_i:     0xE4;  // 11100100 = call %I
  map callr_j:     0xE5;  // 11100101 = call %J
  map call:        0xE6;  // 11100110 = call #ref
```

```
  operands {
    %1 = { +0=0xE4; +0[0:0]=%1; };
    #1 = { +0=0xE6; +1=#1[31:24]; +2=#1[23:16]; +3=#1[15:8];
           +4=#1[7:0]; };
  };
};
op return {
  map return:     0xE7;  // 11100111 = return
  operands { - = { +0=0xE7; }; };
};
op jump {
  map jump_c_0:   0xE8;  // 11101000 = jump %carry,0,#ref
  map jump_c_1:   0xE9;  // 11101001 = jump %carry,1,#ref
  map jump_z_0:   0xEA;  // 11101010 = jump %zero,0,#ref
  map jump_z_1:   0xEB;  // 11101011 = jump %zero,1,#ref
  map jump_n_0:   0xEC;  // 11101100 = jump %negative,0,#ref
  map jump_n_1:   0xED;  // 11101101 = jump %negative,1,#ref
  map jump_v_0:   0xEE;  // 11101110 = jump %overflow,0,#ref
  map jump_v_1:   0xEF;  // 11101111 = jump %overflow,1,#ref
  map jumpr_i:    0xF0;  // 11110000 = jump %I
  map jumpr_j:    0xF1;  // 11110001 = jump %J
  map jump:       0xFC;  // 11111100 = jump #ref
  map jump_cs:    0xFD;  // 11111101 = jump cs0, cs1, #ref
  map jump_ds:    0xFE;  // 11111110 = jump ds0, ds1
  operands {
    %1       = { +0=0xF0; +0[0:0]=%1; };
    %1,#2,#3 = { +0=0xE8; +0[2:1]=%1; +0[0:0]=#2;
                 +1=#3[31:24]; +2=#3[23:16]; +3=#3[15:8]; +4=#3[7:0]; };
    #1       = { +0=0xFC;
                 +1=#1[31:24]; +2=#1[23:16]; +3=#1[15:8]; +4=#1[7:0]; };
    #1,#2,#3 = { +0=0xFD;
                 +1=#1[31:24]; +2=#1[23:16]; +3=#1[15:8]; +4=#1[7:0];
                 +5=#2[31:24]; +6=#2[23:16]; +7=#2[15:8]; +8=#2[7:0];
                 +9=#3[31:24]; +10=#3[23:16]; +11=#3[15:8];
                 +12=#3[7:0]; };
    #1,#2    = { +0=0xFE;
                 +1=#1[31:24]; +2=#1[23:16]; +3=#1[15:8]; +4=#1[7:0];
                 +5=#2[31:24]; +6=#2[23:16]; +7=#2[15:8]; +8=#2[7:0]; };
  };
};
```

```
op imrl {
  map imrl:       0xF2;  // 11110010 = imrl n = IRQ mask register load
  operands {
    #1 = { +0=0xF2; +1=#1[7:0]; };
  };
};
op ivtl {
  map ivtl:       0xF3;  // 11110011 = ivtl #ref = load INT vector table
  operands {
    #1 = { +0=0xF3;
           +1=#1[31:24]; +2=#1[23:16]; +3=#1[15:8]; +4=#1[7:0]; };
  };
};
op int {
  map int:        0xF4;  // 11110100 = int n = soft interrupt call
  operands {
    #1 = { +0=0xF4; +1=#1[7:0]; };
  };
};
op iret {
  map iret:       0xF5;  // 11110101 = iret = interrupt return
  operands { - = { +0=0xF5; }; };
};
op in {
  map in_a:       0xF6;  // 11110110 = in %A,n = A <- I/O device
  map in_b:       0xF7;  // 11110111 = in %B,n = B <- I/O device
  operands {
    %1,#2 = { +0=0xF6; +0[0:0]=%1; +1=#2[7:0]; };
  };
};
op out {
  map out_a:      0xF8;  // 11111000 = out %A,n = I/O device <- A
  map out_b:      0xF9;  // 11111001 = out %B,n = I/O device <- B
  operands {
    %1,#2 = { +0=0xF8; +0[0:0]=%1; +1=#2[7:0]; };
  };
};
op memcpy {
  map memcpy:     0xFA;  // 11111010 = memcpy #src,#dst,size
  operands {
```

```
    #1,#2,#3 = { +0=0xFA;
                 +1=#1[31:24]; +2=#1[23:16]; +3=#1[15:8]; +4=#1[7:0];
                 +5=#2[31:24]; +6=#2[23:16]; +7=#2[15:8]; +8=#2[7:0];
                 +9=#3[31:24]; +10=#3[23:16]; +11=#3[15:8]; +12=#3[7:0];
               };
  };
};
op store_ssp {
  map st_a_ssp: 0xFB;  // 11111011 = SSP <- A
  operands {
    - = { +0=0xFB; };
  };
};
op halt {
  map halt:       0xFF;  // 11111111 = halt
  operands { - = { +0=0xFF; }; };
};
```

13.21 Data initialization

This version of the simple CPU accesses the memory in big-endian mode. As the TKGate compiler initializes variables in little-endian mode, it is necessary to define some instructions that do the same, but in the right byte order. These instructions are declared as if they were opcodes, but actually there is no opcode at all.

Listing 13.60. Variable definition pseudo-opcodes for TKGate.

```
op int8 {
  operands { #1 = { +0[7:0]=#1[7:0]; }; };
};
op int16 {
  operands { #1 = { +0[7:0]=#1[15:8]; +1[7:0]=#1[7:0]; }; };
};
op int24 {
  operands { #1 = { +0[7:0]=#1[23:16]; +1[7:0]=#1[15:8];
                    +2[7:0]=#1[7:0]; }; };
};
op int32 {
  operands { #1 = { +0[7:0]=#1[31:24]; +1[7:0]=#1[23:16];
                    +2[7:0]=#1[15:8]; +3[7:0]=#1[7:0]; }; };
```

```
};
```

13.22 Microcode

It follows the microcode listing, describing the opcode procedures.

Listing 13.61. Microcode for TKGate.

```
begin microcode @ 0
//
fetch:
    // IR <- M[PC++], load
    M=rank8 M=byte0 M=code M=ram_oe M=ir_load PCi=1 CTRL=load;
nop:
    // Just jump to the fetch microcode:
                                        ADDR=fetch JUMP=unconditional;
st_a_b:
    Ar=enable  Br=load                  ADDR=fetch JUMP=unconditional;
st_a_c:
    Ar=enable  Cr=load                  ADDR=fetch JUMP=unconditional;
st_a_d:
    Ar=enable  Dr=load                  ADDR=fetch JUMP=unconditional;
st_a_i:
    Ar=enable  Ir=load                  ADDR=fetch JUMP=unconditional;
st_a_j:
    Ar=enable  Jr=load                  ADDR=fetch JUMP=unconditional;
st_a_sp:
    Ar=enable  SPr=load                 ADDR=fetch JUMP=unconditional;
st_a_fp:
    Ar=enable  FPr=load                 ADDR=fetch JUMP=unconditional;
st_b_a:
    Br=enable  Ar=load                  ADDR=fetch JUMP=unconditional;
st_b_c:
    Br=enable  Cr=load                  ADDR=fetch JUMP=unconditional;
st_b_d:
    Br=enable  Dr=load                  ADDR=fetch JUMP=unconditional;
st_b_i:
    Br=enable  Ir=load                  ADDR=fetch JUMP=unconditional;
st_b_j:
    Br=enable  Jr=load                  ADDR=fetch JUMP=unconditional;
```

```
st_b_sp:
    Br=enable  SPr=load                 ADDR=fetch JUMP=unconditional;
st_b_fp:
    Br=enable  FPr=load                 ADDR=fetch JUMP=unconditional;
ld_a_b:
    Br=enable  Ar=load                  ADDR=fetch JUMP=unconditional;
ld_a_c:
    Cr=enable  Ar=load                  ADDR=fetch JUMP=unconditional;
ld_a_d:
    Dr=enable  Ar=load                  ADDR=fetch JUMP=unconditional;
ld_a_i:
    Ir=enable  Ar=load                  ADDR=fetch JUMP=unconditional;
ld_a_j:
    Jr=enable  Ar=load                  ADDR=fetch JUMP=unconditional;
ld_a_sp:
    SPr=enable Ar=load                  ADDR=fetch JUMP=unconditional;
ld_a_fp:
    FPr=enable Ar=load                  ADDR=fetch JUMP=unconditional;
ld_b_a:
    Br=enable  Br=load                  ADDR=fetch JUMP=unconditional;
ld_b_c:
    Cr=enable  Br=load                  ADDR=fetch JUMP=unconditional;
ld_b_d:
    Dr=enable  Br=load                  ADDR=fetch JUMP=unconditional;
ld_b_i:
    Ir=enable  Br=load                  ADDR=fetch JUMP=unconditional;
ld_b_j:
    Jr=enable  Br=load                  ADDR=fetch JUMP=unconditional;
ld_b_sp:
    SPr=enable Br=load                  ADDR=fetch JUMP=unconditional;
ld_b_fp:
    FPr=enable Br=load                  ADDR=fetch JUMP=unconditional;
ld8ur_i:
    DA=i  M=rank8 M=byte0 M=data M=ram_oe M=unsigned M=md_enable
        Ar=load                         ADDR=fetch JUMP=unconditional;
ld8ur_j:
    DA=j  M=rank8 M=byte0 M=data M=ram_oe M=unsigned M=md_enable
        Ar=load                         ADDR=fetch JUMP=unconditional;
ld8ur_sp:
    DA=sp M=rank8 M=byte0 M=data M=ram_oe M=unsigned M=md_enable
```

```
        Ar=load                          ADDR=fetch JUMP=unconditional;
ld8ur_fp:
    DA=fp M=rank8 M=byte0 M=data M=ram_oe M=unsigned M=md_enable
        Ar=load                          ADDR=fetch JUMP=unconditional;
ld8sr_i:
    DA=i  M=rank8 M=byte0 M=data M=ram_oe M=signed M=md_enable
        Ar=load                          ADDR=fetch JUMP=unconditional;
ld8sr_j:
    DA=j  M=rank8 M=byte0 M=data M=ram_oe M=signed M=md_enable
        Ar=load                          ADDR=fetch JUMP=unconditional;
ld8sr_sp:
    DA=sp M=rank8 M=byte0 M=data M=ram_oe M=signed M=md_enable
        Ar=load                          ADDR=fetch JUMP=unconditional;
ld8sr_fp:
    DA=fp M=rank8 M=byte0 M=data M=ram_oe M=signed M=md_enable
        Ar=load                          ADDR=fetch JUMP=unconditional;
ld8u:
    M=rank8 M=byte0 M=code M=ram_oe M=unsigned M=md_enable
        Ar=load PCi=1                    ADDR=fetch JUMP=unconditional;
ld8s:
    M=rank8 M=byte0 M=code M=ram_oe M=signed M=md_enable
        Ar=load PCi=1                    ADDR=fetch JUMP=unconditional;
ld16ur_i:
    DA=i  M=rank16 M=byte1 M=data M=ram_oe;
    DA=i  M=rank16 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                            ADDR=fetch JUMP=unconditional;
ld16ur_j:
    DA=j  M=rank16 M=byte1 M=data M=ram_oe;
    DA=j  M=rank16 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                            ADDR=fetch JUMP=unconditional;
ld16ur_sp:
    DA=sp M=rank16 M=byte1 M=data M=ram_oe;
    DA=sp M=rank16 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                            ADDR=fetch JUMP=unconditional;
ld16ur_fp:
    DA=fp M=rank16 M=byte1 M=data M=ram_oe;
    DA=fp M=rank16 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                            ADDR=fetch JUMP=unconditional;
ld16sr_i:
    DA=i  M=rank16 M=byte1 M=data M=ram_oe;
```

```
    DA=i  M=rank16 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                          ADDR=fetch JUMP=unconditional;
ld16sr_j:
    DA=j  M=rank16 M=byte1 M=data M=ram_oe;
    DA=j  M=rank16 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                          ADDR=fetch JUMP=unconditional;
ld16sr_sp:
    DA=sp M=rank16 M=byte1 M=data M=ram_oe;
    DA=sp M=rank16 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                          ADDR=fetch JUMP=unconditional;
ld16sr_fp:
    DA=fp M=rank16 M=byte1 M=data M=ram_oe;
    DA=fp M=rank16 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                          ADDR=fetch JUMP=unconditional;
ld16u:
    M=rank16 M=byte1 M=code M=ram_oe;
    M=rank16 M=byte0 M=code M=ram_oe M=md_enable M=unsigned Ar=load
      PCi=2             ADDR=fetch JUMP=unconditional;
ld16s:
    M=rank16 M=byte1 M=code M=ram_oe;
    M=rank16 M=byte0 M=code M=ram_oe M=md_enable M=signed   Ar=load
      PCi=2                            ADDR=fetch JUMP=unconditional;
ld24ur_i:
    DA=i  M=rank24 M=byte2 M=data M=ram_oe;
    DA=i  M=rank24 M=byte1 M=data M=ram_oe;
    DA=i  M=rank24 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                          ADDR=fetch JUMP=unconditional;
ld24ur_j:
    DA=j  M=rank24 M=byte2 M=data M=ram_oe;
    DA=j  M=rank24 M=byte1 M=data M=ram_oe;
    DA=j  M=rank24 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                          ADDR=fetch JUMP=unconditional;
ld24ur_sp:
    DA=sp M=rank24 M=byte2 M=data M=ram_oe;
    DA=sp M=rank24 M=byte1 M=data M=ram_oe;
    DA=sp M=rank24 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                          ADDR=fetch JUMP=unconditional;
ld24ur_fp:
    DA=fp M=rank24 M=byte2 M=data M=ram_oe;
    DA=fp M=rank24 M=byte1 M=data M=ram_oe;
```

```
    DA=fp M=rank24 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                           ADDR=fetch JUMP=unconditional;
ld24sr_i:
    DA=i  M=rank24 M=byte2 M=data M=ram_oe;
    DA=i  M=rank24 M=byte1 M=data M=ram_oe;
    DA=i  M=rank24 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                           ADDR=fetch JUMP=unconditional;
ld24sr_j:
    DA=j  M=rank24 M=byte2 M=data M=ram_oe;
    DA=j  M=rank24 M=byte1 M=data M=ram_oe;
    DA=j  M=rank24 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                           ADDR=fetch JUMP=unconditional;
ld24sr_sp:
    DA=sp M=rank24 M=byte2 M=data M=ram_oe;
    DA=sp M=rank24 M=byte1 M=data M=ram_oe;
    DA=sp M=rank24 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                           ADDR=fetch JUMP=unconditional;
ld24sr_fp:
    DA=fp M=rank24 M=byte2 M=data M=ram_oe;
    DA=fp M=rank24 M=byte1 M=data M=ram_oe;
    DA=fp M=rank24 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                           ADDR=fetch JUMP=unconditional;
ld24u:
    M=rank24 M=byte2 M=code M=ram_oe;
    M=rank24 M=byte1 M=code M=ram_oe;
    M=rank24 M=byte0 M=code M=ram_oe M=md_enable M=unsigned
      Ar=load PCi=3                     ADDR=fetch JUMP=unconditional;
ld24s:
    M=rank24 M=byte2 M=code M=ram_oe;
    M=rank24 M=byte1 M=code M=ram_oe;
    M=rank24 M=byte0 M=code M=ram_oe M=md_enable M=signed
      Ar=load PCi=3                     ADDR=fetch JUMP=unconditional;
ld32u:
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=unsigned
      Ar=load PCi=4                     ADDR=fetch JUMP=unconditional;
ld32s:
    M=rank32 M=byte3 M=code M=ram_oe;
```

```
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=signed
      Ar=load PCi=4                       ADDR=fetch JUMP=unconditional;
ld32ur_i:
    DA=i  M=rank32 M=byte3 M=data M=ram_oe;
    DA=i  M=rank32 M=byte2 M=data M=ram_oe;
    DA=i  M=rank32 M=byte1 M=data M=ram_oe;
    DA=i  M=rank32 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                             ADDR=fetch JUMP=unconditional;
ld32ur_j:
    DA=j  M=rank32 M=byte3 M=data M=ram_oe;
    DA=j  M=rank32 M=byte2 M=data M=ram_oe;
    DA=j  M=rank32 M=byte1 M=data M=ram_oe;
    DA=j  M=rank32 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                             ADDR=fetch JUMP=unconditional;
ld32ur_sp:
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                             ADDR=fetch JUMP=unconditional;
ld32ur_fp:
    DA=fp M=rank32 M=byte3 M=data M=ram_oe;
    DA=fp M=rank32 M=byte2 M=data M=ram_oe;
    DA=fp M=rank32 M=byte1 M=data M=ram_oe;
    DA=fp M=rank32 M=byte0 M=data M=ram_oe M=md_enable M=unsigned
      Ar=load                             ADDR=fetch JUMP=unconditional;
ld32sr_i:
    DA=i  M=rank32 M=byte3 M=data M=ram_oe;
    DA=i  M=rank32 M=byte2 M=data M=ram_oe;
    DA=i  M=rank32 M=byte1 M=data M=ram_oe;
    DA=i  M=rank32 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                             ADDR=fetch JUMP=unconditional;
ld32sr_j:
    DA=j  M=rank32 M=byte3 M=data M=ram_oe;
    DA=j  M=rank32 M=byte2 M=data M=ram_oe;
    DA=j  M=rank32 M=byte1 M=data M=ram_oe;
    DA=j  M=rank32 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                             ADDR=fetch JUMP=unconditional;
```

```
ld32sr_sp:
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                          ADDR=fetch JUMP=unconditional;
ld32sr_fp:
    DA=fp M=rank32 M=byte3 M=data M=ram_oe;
    DA=fp M=rank32 M=byte2 M=data M=ram_oe;
    DA=fp M=rank32 M=byte1 M=data M=ram_oe;
    DA=fp M=rank32 M=byte0 M=data M=ram_oe M=md_enable M=signed
      Ar=load                          ADDR=fetch JUMP=unconditional;
lea_a_i:
    DA=i  DA=enable Ar=load;           ADDR=fetch JUMP=unconditional;
lea_a_j:
    DA=j  DA=enable Ar=load            ADDR=fetch JUMP=unconditional;
lea_a_sp:
    DA=sp DA=enable Ar=load            ADDR=fetch JUMP=unconditional;
lea_a_fp:
    DA=fp DA=enable Ar=load            ADDR=fetch JUMP=unconditional;
lea_b_i:
    DA=i  DA=enable Br=load            ADDR=fetch JUMP=unconditional;
lea_b_j:
    DA=j  DA=enable Br=load            ADDR=fetch JUMP=unconditional;
lea_b_sp:
    DA=sp DA=enable Br=load            ADDR=fetch JUMP=unconditional;
lea_b_fp:
    DA=fp DA=enable Br=load            ADDR=fetch JUMP=unconditional;
swap_a_a:
                                       ADDR=fetch JUMP=unconditional;
swap_b_b:
                                       ADDR=fetch JUMP=unconditional;
swap_b_a:
    Ar=enable    TMP0=load;
    Br=enable    Ar=load;
    TMP0=enable  Br=load               ADDR=fetch JUMP=unconditional;
swap_a_b:
    Ar=enable    TMP0=load;
    Br=enable    Ar=load;
    TMP0=enable  Br=load               ADDR=fetch JUMP=unconditional;
```

```
swap_a_c:
    Ar=enable    TMP0=load;
    Cr=enable    Ar=load;
    TMP0=enable  Cr=load                ADDR=fetch JUMP=unconditional;
swap_a_d:
    Ar=enable    TMP0=load;
    Dr=enable    Ar=load;
    TMP0=enable  Dr=load                ADDR=fetch JUMP=unconditional;
swap_a_i:
    Ar=enable    TMP0=load;
    Ir=enable    Ar=load;
    TMP0=enable  Ir=load                ADDR=fetch JUMP=unconditional;
swap_a_j:
    Ar=enable    TMP0=load;
    Jr=enable    Ar=load;
    TMP0=enable  Jr=load                ADDR=fetch JUMP=unconditional;
swap_a_sp:
    Ar=enable    TMP0=load;
    SPr=enable   Ar=load;
    TMP0=enable  SPr=load               ADDR=fetch JUMP=unconditional;
swap_a_fp:
    Ar=enable    TMP0=load;
    FPr=enable   Ar=load;
    TMP0=enable  FPr=load               ADDR=fetch JUMP=unconditional;
swap_b_c:
    Br=enable    TMP0=load;
    Cr=enable    Br=load;
    TMP0=enable  Cr=load                ADDR=fetch JUMP=unconditional;
swap_b_d:
    Br=enable    TMP0=load;
    Dr=enable    Br=load;
    TMP0=enable  Dr=load                ADDR=fetch JUMP=unconditional;
swap_b_i:
    Br=enable    TMP0=load;
    Ir=enable    Br=load;
    TMP0=enable  Ir=load                ADDR=fetch JUMP=unconditional;
swap_b_j:
    Br=enable    TMP0=load;
    Jr=enable    Br=load;
    TMP0=enable  Jr=load                ADDR=fetch JUMP=unconditional;
```

```
swap_b_sp:
    Br=enable    TMP0=load;
    SPr=enable   Br=load;
    TMP0=enable  SPr=load                ADDR=fetch JUMP=unconditional;
swap_b_fp:
    Br=enable    TMP0=load;
    FPr=enable   Br=load;
    TMP0=enable  FPr=load                ADDR=fetch JUMP=unconditional;


st8r_i:
    Ar=enable DA=i  M=rank8  M=byte0 M=data M=ram_we
                                         ADDR=fetch JUMP=unconditional;
st8r_j:
    Ar=enable DA=j  M=rank8  M=byte0 M=data M=ram_we
                                         ADDR=fetch JUMP=unconditional;
st8r_sp:
    Ar=enable DA=sp M=rank8  M=byte0 M=data M=ram_we
                                         ADDR=fetch JUMP=unconditional;
st8r_fp:
    Ar=enable DA=fp M=rank8  M=byte0 M=data M=ram_we
                                         ADDR=fetch JUMP=unconditional;
st16r_i:
    Ar=enable DA=i  M=rank16 M=byte1 M=data M=ram_we;
    Ar=enable DA=i  M=rank16 M=byte0 M=data M=ram_we
                                         ADDR=fetch JUMP=unconditional;
st16r_j:
    Ar=enable DA=j  M=rank16 M=byte1 M=data M=ram_we;
    Ar=enable DA=j  M=rank16 M=byte0 M=data M=ram_we
                                         ADDR=fetch JUMP=unconditional;
st16r_sp:
    Ar=enable DA=sp M=rank16 M=byte1 M=data M=ram_we;
    Ar=enable DA=sp M=rank16 M=byte0 M=data M=ram_we
                                         ADDR=fetch JUMP=unconditional;
st16r_fp:
    Ar=enable DA=fp M=rank16 M=byte1 M=data M=ram_we;
    Ar=enable DA=fp M=rank16 M=byte0 M=data M=ram_we
                                         ADDR=fetch JUMP=unconditional;
st24r_i:
    Ar=enable DA=i  M=rank24 M=byte2 M=data M=ram_we;
```

```
    Ar=enable DA=i  M=rank24 M=byte1 M=data M=ram_we;
    Ar=enable DA=i  M=rank24 M=byte0 M=data M=ram_we
                                        ADDR=fetch JUMP=unconditional;
st24r_j:
    Ar=enable DA=j  M=rank24 M=byte2 M=data M=ram_we;
    Ar=enable DA=j  M=rank24 M=byte1 M=data M=ram_we;
    Ar=enable DA=j  M=rank24 M=byte0 M=data M=ram_we
                                        ADDR=fetch JUMP=unconditional;
st24r_sp:
    Ar=enable DA=sp M=rank24 M=byte2 M=data M=ram_we;
    Ar=enable DA=sp M=rank24 M=byte1 M=data M=ram_we;
    Ar=enable DA=sp M=rank24 M=byte0 M=data M=ram_we
                                        ADDR=fetch JUMP=unconditional;
st24r_fp:
    Ar=enable DA=fp M=rank24 M=byte2 M=data M=ram_we;
    Ar=enable DA=fp M=rank24 M=byte1 M=data M=ram_we;
    Ar=enable DA=fp M=rank24 M=byte0 M=data M=ram_we
                                        ADDR=fetch JUMP=unconditional;
st32r_i:
    Ar=enable DA=i  M=rank32 M=byte3 M=data M=ram_we;
    Ar=enable DA=i  M=rank32 M=byte2 M=data M=ram_we;
    Ar=enable DA=i  M=rank32 M=byte1 M=data M=ram_we;
    Ar=enable DA=i  M=rank32 M=byte0 M=data M=ram_we
                                        ADDR=fetch JUMP=unconditional;
st32r_j:
    Ar=enable DA=j  M=rank32 M=byte3 M=data M=ram_we;
    Ar=enable DA=j  M=rank32 M=byte2 M=data M=ram_we;
    Ar=enable DA=j  M=rank32 M=byte1 M=data M=ram_we;
    Ar=enable DA=j  M=rank32 M=byte0 M=data M=ram_we
                                        ADDR=fetch JUMP=unconditional;
st32r_sp:
    Ar=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    Ar=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    Ar=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    Ar=enable DA=sp M=rank32 M=byte0 M=data M=ram_we
                                        ADDR=fetch JUMP=unconditional;
st32r_fp:
    Ar=enable DA=fp M=rank32 M=byte3 M=data M=ram_we;
    Ar=enable DA=fp M=rank32 M=byte2 M=data M=ram_we;
    Ar=enable DA=fp M=rank32 M=byte1 M=data M=ram_we;
```

```
    Ar=enable DA=fp M=rank32 M=byte0 M=data M=ram_we
                                        ADDR=fetch JUMP=unconditional;
push_a:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load SPi=-4;
    Ar=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    Ar=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    Ar=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    Ar=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    TMP0=enable Jr=load                 ADDR=fetch JUMP=unconditional;
push_b:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0 CNS=enable Jr=load SPi=-4;
    Br=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    Br=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    Br=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    Br=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    TMP0=enable Jr=load                 ADDR=fetch JUMP=unconditional;
push_c:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0 CNS=enable Jr=load SPi=-4;
    Cr=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    Cr=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    Cr=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    Cr=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    TMP0=enable Jr=load                 ADDR=fetch JUMP=unconditional;
push_d:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0 CNS=enable Jr=load SPi=-4;
    Dr=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    Dr=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    Dr=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    Dr=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    TMP0=enable Jr=load                 ADDR=fetch JUMP=unconditional;
push_i:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0 CNS=enable Jr=load SPi=-4;
    Ir=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    Ir=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    Ir=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
```

```
    Ir=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
push_j:
    Jr=enable TMP0=load;          // %J is saved and later restored.
    CNSv=0 CNS=enable Jr=load SPi=-4;
    TMP0=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    TMP0=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    TMP0=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    TMP0=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
push_sp:
    Jr=enable TMP0=load;          // %J is saved and later restored.
    CNSv=-4 CNS=enable Jr=load;
    SPr=enable TMP1=load SPi=-4;
    TMP1=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    TMP1=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    TMP1=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    TMP1=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
push_fp:
    Jr=enable TMP0=load;          // %J is saved and later restored.
    CNSv=0 CNS=enable Jr=load SPi=-4;
    FPr=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    FPr=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    FPr=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    FPr=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
pop_a:
    Jr=enable TMP0=load;          // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable Ar=load SPi=4;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
pop_b:
    Jr=enable TMP0=load;          // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
```

```
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable Br=load SPi=4;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
pop_c:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable Cr=load SPi=4;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
pop_d:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable Dr=load SPi=4;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
pop_i:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable Ir=load SPi=4;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
pop_j:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable Jr=load SPi=4
                                       ADDR=fetch JUMP=unconditional;
pop_sp:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
```

```
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable SPr=load;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
pop_fp:
    Jr=enable TMP0=load;        // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable FPr=load SPi=4;
    TMP0=enable Jr=load                ADDR=fetch JUMP=unconditional;
equal8s:
    ALU=rank8  ALU=equal ALU=signed FL=load_alu ALU=enable Ar=load
                                       ADDR=fetch JUMP=unconditional;
equal16s:
    ALU=rank16 ALU=equal ALU=signed FL=load_alu ALU=enable Ar=load
                                       ADDR=fetch JUMP=unconditional;
equal24s:
    ALU=rank24 ALU=equal ALU=signed FL=load_alu ALU=enable Ar=load
                                       ADDR=fetch JUMP=unconditional;
equal32s:
    ALU=rank32 ALU=equal ALU=signed FL=load_alu ALU=enable Ar=load
                                       ADDR=fetch JUMP=unconditional;
not8s:
    ALU=rank8  ALU=not   ALU=signed FL=load_alu ALU=enable Ar=load
                                       ADDR=fetch JUMP=unconditional;
not16s:
    ALU=rank16 ALU=not   ALU=signed FL=load_alu ALU=enable Ar=load
                                       ADDR=fetch JUMP=unconditional;
not24s:
    ALU=rank24 ALU=not   ALU=signed FL=load_alu ALU=enable Ar=load
                                       ADDR=fetch JUMP=unconditional;
not32s:
    ALU=rank32 ALU=not   ALU=signed FL=load_alu ALU=enable Ar=load
                                       ADDR=fetch JUMP=unconditional;
and8s:
    ALU=rank8  ALU=and   ALU=signed FL=load_alu ALU=enable Ar=load
                                       ADDR=fetch JUMP=unconditional;
and16s:
    ALU=rank16 ALU=and   ALU=signed FL=load_alu ALU=enable Ar=load
```

```
                                        ADDR=fetch JUMP=unconditional;
and24s:
    ALU=rank24 ALU=and   ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
and32s:
    ALU=rank32 ALU=and   ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
nand8s:
    ALU=rank8  ALU=nand  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
nand16s:
    ALU=rank16 ALU=nand  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
nand24s:
    ALU=rank24 ALU=nand  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
nand32s:
    ALU=rank32 ALU=nand  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
or8s:
    ALU=rank8  ALU=or    ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
or16s:
    ALU=rank16 ALU=or    ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
or24s:
    ALU=rank24 ALU=or    ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
or32s:
    ALU=rank32 ALU=or    ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
nor8s:
    ALU=rank8  ALU=nor   ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
nor16s:
    ALU=rank16 ALU=nor   ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
nor24s:
    ALU=rank24 ALU=nor   ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
```

```
nor32s:
    ALU=rank32 ALU=nor   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
xor8s:
    ALU=rank8  ALU=xor   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
xor16s:
    ALU=rank16 ALU=xor   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
xor24s:
    ALU=rank24 ALU=xor   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
xor32s:
    ALU=rank32 ALU=xor   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
nxor8s:
    ALU=rank8  ALU=nxor  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
nxor16s:
    ALU=rank16 ALU=nxor  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
nxor24s:
    ALU=rank24 ALU=nxor  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
nxor32s:
    ALU=rank32 ALU=nxor  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
add8s:
    ALU=rank8  ALU=add   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
add16s:
    ALU=rank16 ALU=add   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
add24s:
    ALU=rank24 ALU=add   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
add32s:
    ALU=rank32 ALU=add   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
sub8s:
```

```
    ALU=rank8  ALU=sub   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
sub16s:
    ALU=rank16 ALU=sub   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
sub24s:
    ALU=rank24 ALU=sub   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
sub32s:
    ALU=rank32 ALU=sub   ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
addc8s:
    ALU=rank8  ALU=addc  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
addc16s:
    ALU=rank16 ALU=addc  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
addc24s:
    ALU=rank24 ALU=addc  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
addc32s:
    ALU=rank32 ALU=addc  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
subb8s:
    ALU=rank8  ALU=subb  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
subb16s:
    ALU=rank16 ALU=subb  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
subb24s:
    ALU=rank24 ALU=subb  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
subb32s:
    ALU=rank32 ALU=subb  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
lshl8s:
    ALU=rank8  ALU=lshl  ALU=signed FL=load_alu ALU=enable Ar=load
                                          ADDR=fetch JUMP=unconditional;
lshl16s:
    ALU=rank16 ALU=lshl  ALU=signed FL=load_alu ALU=enable Ar=load
```

```
                                        ADDR=fetch JUMP=unconditional;
lshl24s:
    ALU=rank24 ALU=lshl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
lshl32s:
    ALU=rank32 ALU=lshl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
lshr8s:
    ALU=rank8  ALU=lshr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
lshr16s:
    ALU=rank16 ALU=lshr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
lshr24s:
    ALU=rank24 ALU=lshr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
lshr32s:
    ALU=rank32 ALU=lshr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
ashl8s:
    ALU=rank8  ALU=ashl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
ashl16s:
    ALU=rank16 ALU=ashl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
ashl24s:
    ALU=rank24 ALU=ashl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
ashl32s:
    ALU=rank32 ALU=ashl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
ashr8s:
    ALU=rank8  ALU=ashr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
ashr16s:
    ALU=rank16 ALU=ashr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
ashr24s:
    ALU=rank24 ALU=ashr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
```

```
ashr32s:
    ALU=rank32 ALU=ashr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotl8s:
    ALU=rank8  ALU=rotl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotl16s:
    ALU=rank16 ALU=rotl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotl24s:
    ALU=rank24 ALU=rotl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotl32s:
    ALU=rank32 ALU=rotl  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotr8s:
    ALU=rank8  ALU=rotr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotr16s:
    ALU=rank16 ALU=rotr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotr24s:
    ALU=rank24 ALU=rotr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotr32s:
    ALU=rank32 ALU=rotr  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotcl8s:
    ALU=rank8  ALU=lshcl ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotcl16s:
    ALU=rank16 ALU=lshcl ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotcl24s:
    ALU=rank24 ALU=lshcl ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotcl32s:
    ALU=rank32 ALU=lshcl ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotcr8s:
```

```
    ALU=rank8  ALU=lshcr ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotcr16s:
    ALU=rank16 ALU=lshcr ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotcr24s:
    ALU=rank24 ALU=lshcr ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
rotcr32s:
    ALU=rank32 ALU=lshcr ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
utrunc8:
    ALU=rank8  ALU=equal ALU=unsigned FL=load_alu ALU=enable
        Ar=load                         ADDR=fetch JUMP=unconditional;
utrunc16:
    ALU=rank16 ALU=equal ALU=unsigned FL=load_alu ALU=enable
        Ar=load                         ADDR=fetch JUMP=unconditional;
utrunc24:
    ALU=rank24 ALU=equal ALU=unsigned FL=load_alu ALU=enable
        Ar=load                         ADDR=fetch JUMP=unconditional;
utrunc32:
    ALU=rank32 ALU=equal ALU=unsigned FL=load_alu ALU=enable
        Ar=load                         ADDR=fetch JUMP=unconditional;
strunc8:
    ALU=rank8  ALU=equal ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
strunc16:
    ALU=rank16 ALU=equal ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
strunc24:
    ALU=rank24 ALU=equal ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
strunc32:
    ALU=rank32 ALU=equal ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
clear_a:
    ALU=rank32 ALU=set0  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
clear_b:
    ALU=rank32 ALU=set0  ALU=signed FL=load_alu ALU=enable Br=load
```

```
                                        ADDR=fetch JUMP=unconditional;
set_a:
    ALU=rank32 ALU=set1  ALU=signed FL=load_alu ALU=enable Ar=load
                                        ADDR=fetch JUMP=unconditional;
set_b:
    ALU=rank32 ALU=set1  ALU=signed FL=load_alu ALU=enable Br=load
                                        ADDR=fetch JUMP=unconditional;
comp:
    ALU=rank32 ALU=and   FL=load_alu
                                        ADDR=fetch JUMP=unconditional;
test:
    ALU=rank32 ALU=sub   FL=load_alu
                                        ADDR=fetch JUMP=unconditional;
flag_c_0:
    ALU=rank32 ALU=carry0 FL=load_alu
                                        ADDR=fetch JUMP=unconditional;
flag_c_1:
    ALU=rank32 ALU=carry1 FL=load_alu
                                        ADDR=fetch JUMP=unconditional;
flag_i_0:
    // Only a supervisor process can change the IRQ enable flag.
    ADDR=fetch JUMP=conditional JUMP=supervisor_f;
    // It is a supervisor process, so the flag is changed.
    ALU=rank32 ALU=irqen0 FL=load_alu
                                        ADDR=fetch JUMP=unconditional;
flag_i_1:
    // Only a supervisor process can change the IRQ enable flag.
    ADDR=fetch JUMP=conditional JUMP=supervisor_f;
    // It is a supervisor process, so the flag is changed.
    ALU=rank32 ALU=irqen1 FL=load_alu
                                        ADDR=fetch JUMP=unconditional;
flag_t_0:
    ALU=rank32 ALU=transition0 FL=load_alu
                                        ADDR=fetch JUMP=unconditional;
flag_t_1:
    // Only a supervisor process can change the IRQ enable flag.
    ADDR=fetch JUMP=conditional JUMP=supervisor_f;
    // It is a supervisor process, so the flag is changed.
    ALU=rank32 ALU=transition1 FL=load_alu
                                        ADDR=fetch JUMP=unconditional;
```

```
call:
    //
    // Push PC+4 upon the stack.
    //
    PCi=4 Jr=enable TMP0=load;
    CNSv=0 CNS=enable Jr=load SPi=-4;
    PCr=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte0 M=data M=ram_we PCi=-4;
    TMP0=enable Jr=load;
    //
    // Put the destination address inside the PC register.
    //
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=unsigned PCr=load
                                        ADDR=fetch JUMP=unconditional;
callr_i:
    //
    // Push PC upon the stack.
    //
    Jr=enable TMP0=load;
    CNSv=0 CNS=enable Jr=load SPi=-4;
    PCr=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    TMP0=enable Jr=load;
    //
    // PC <- I
    //
    Ir=enable PCr=load                  ADDR=fetch JUMP=unconditional;
callr_j:
    //
    // Push PC upon the stack.
    //
    SPi=-4;
    PCr=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
```

```
    PCr=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    //
    // PC <- J
    //
    Jr=enable PCr=load                  ADDR=fetch JUMP=unconditional;
return:
    Jr=enable TMP0=load;          // %J is saved and later restored.
    CNSv=0x00 CNS=enable Jr=load;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable PCr=load SPi=4;
    TMP0=enable Jr=load                 ADDR=fetch JUMP=unconditional;
jump:
    //
    // Load the destination address inside the PC register.
    //
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=unsigned PCr=load
                                        ADDR=fetch JUMP=unconditional;
jumpr_i:
    Ir=enable PCr=load                  ADDR=fetch JUMP=unconditional;
jumpr_j:
    Jr=enable PCr=load                  ADDR=fetch JUMP=unconditional;
jump_c_0:
    ADDR=jump JUMP=conditional JUMP=carry_f;    // Jump if...
    PCi=4                               ADDR=fetch JUMP=unconditional;
jump_c_1:
    ADDR=jump JUMP=conditional JUMP=carry_t;    // Jump if...
    PCi=4                               ADDR=fetch JUMP=unconditional;
jump_z_0:
    ADDR=jump JUMP=conditional JUMP=zero_f;     // Jump if...
    PCi=4                               ADDR=fetch JUMP=unconditional;
jump_z_1:
    ADDR=jump JUMP=conditional JUMP=zero_t;     // Jump if...
    PCi=4                               ADDR=fetch JUMP=unconditional;
```

```
jump_n_0:
    ADDR=jump JUMP=conditional JUMP=negative_f; // Jump if...
    PCi=4                                  ADDR=fetch JUMP=unconditional;
jump_n_1:
    ADDR=jump JUMP=conditional JUMP=negative_t; // Jump if...
    PCi=4                                  ADDR=fetch JUMP=unconditional;
jump_v_0:
    ADDR=jump JUMP=conditional JUMP=overflow_f; // Jump if...
    PCi=4                                  ADDR=fetch JUMP=unconditional;
jump_v_1:
    ADDR=jump JUMP=conditional JUMP=overflow_t; // Jump if...
    PCi=4                                  ADDR=fetch JUMP=unconditional;
jump_cs:
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=unsigned TMP0=load
        PCi=4;
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=unsigned TMP1=load
        PCi=4;
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=unsigned PCr=load;
    TMP0=enable CS0=load;
    TMP1=enable CS1=load                   ADDR=fetch JUMP=unconditional;
jump_ds:
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=unsigned DS0=load
        PCi=4;
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=unsigned DS1=load
        PCi=4                              ADDR=fetch JUMP=unconditional;
```

```
imrl:
    M=rank8 M=byte0 M=code M=ram_oe M=md_enable M=unsigned
        IRQ=mask_load PCi=1             ADDR=fetch JUMP=unconditional;
ivtl:
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable M=unsigned IVT=load
        PCi=4                           ADDR=fetch JUMP=unconditional;
irq:
    //
    // Take the PC back to the original opcode, convert
    // the IRQ into a IVT pointer and tell that the IRQ is served;
    // then jump to the J clear phase.
    //
    PCi=-1 IVT=irq IVT=enable TMP0=load IRQ=irq_done
    ADDR=int_j_clear JUMP=unconditional;
    //
int:
    //
    // Get the interrupt number, convert into the pointer inside
    // the IVT table, then save it into TMP0.
    //
    M=rank8 M=byte0 M=code M=ram_oe IVT=int IVT=enable TMP0=load
        PCi=1;
    //
    // Continue with the following code.
    //
int_j_clear:
    //
    // Save and clear J, which is used for memory access (TMP1).
    //
    Jr=enable TMP1=load;
    CNSv=0 CNS=enable Jr=load;
    //
    // Check if the process is already at the supervisor level.
    //
    ADDR=int_supervisor_fl JUMP=conditional JUMP=supervisor_t;
    //
    // It is not already in supervisor mode and the procedure
```

```
    // necessary to gain the supervisor status is entered.
    //
int_user:
    //
    // Save and clear DS0 and DS1, because the supervisor
    // data segment is fixed at DS0=0 and DS1=0 (TMP2 and TMP3).
    //
    DS0=enable TMP2=load;
    DS1=enable TMP3=load;
    CNSv=0 CNS=enable DS0=load DS1=load;
    //
    // Save SP and get the supervisor stack pointer from SSP (TMP4):
    // it is assumed that SSP is correctly set up.
    //
    SPr=enable TMP4=load;
    SSP=enable SPr=load;
    //
    // Now SP is pointing the supervisor stack: user SP,
    // DS1 and DS0 are pushed onto the stack.
    //
    SPi=-4;
    TMP4=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    TMP4=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    TMP4=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    TMP4=enable DA=sp M=rank32 M=byte0 M=data M=ram_we SPi=-4;
    TMP3=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    TMP3=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    TMP3=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    TMP3=enable DA=sp M=rank32 M=byte0 M=data M=ram_we SPi=-4;
    TMP2=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    TMP2=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    TMP2=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    TMP2=enable DA=sp M=rank32 M=byte0 M=data M=ram_we SPi=-4;
    //
    // Now the FL is pushed onto the stack, then the status
    // is changed: 'supervisor' and 'transition' flags are set,
    // whereas the IRQ enable flag is disabled.
    //
    FL=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    FL=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
```

```
    FL=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    FL=enable DA=sp M=rank32 M=byte0 M=data M=ram_we SPi=-4;
    //
    ALU=rank32 ALU=irqen0 FL=load_alu;
    ALU=rank32 ALU=supervisor1 FL=load_alu;
    ALU=rank32 ALU=transition1 FL=load_alu;
    //
    // Continue with the remaining register to be
    // saved.
    //
    ADDR=int_supervisor_cs_pc JUMP=unconditional;
    //
int_supervisor_fl:
    //
    // The FL is pushed onto the stack and the status
    // is changed: IRQ enable flag is disabled.
    //
    SPi=-4;
    FL=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    FL=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    FL=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    FL=enable DA=sp M=rank32 M=byte0 M=data M=ram_we SPi=-4;
    //
    ALU=rank32 ALU=irqen0 FL=load_alu;
    //
    // Continue with the following code.
    //
int_supervisor_cs_pc:
    //
    // The code segment registers and the program counter
    // are saved on to the stack, then the code segment
    // registers are cleared.
    //
    CS1=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    CS1=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    CS1=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    CS1=enable DA=sp M=rank32 M=byte0 M=data M=ram_we SPi=-4;
    CS0=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    CS0=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    CS0=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
```

```
    CS0=enable DA=sp M=rank32 M=byte0 M=data M=ram_we SPi=-4;
    PCr=enable DA=sp M=rank32 M=byte3 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte2 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte1 M=data M=ram_we;
    PCr=enable DA=sp M=rank32 M=byte0 M=data M=ram_we;
    //
    CNSv=0 CNS=enable CS0=load CS1=load;
    //
    // Dereference the IVT pointer into the interrupt
    // function address, loading the new value into the
    // PC register.
    //
    TMP0=enable Jr=load;
    DA=j  M=rank32 M=byte3 M=data M=ram_oe;
    DA=j  M=rank32 M=byte2 M=data M=ram_oe;
    DA=j  M=rank32 M=byte1 M=data M=ram_oe;
    DA=j  M=rank32 M=byte0 M=data M=ram_oe M=md_enable PCr=load;
    //
    // Restore the J register and finish.
    //
    TMP1=enable Jr=load                ADDR=fetch JUMP=unconditional;
//
iret:
    //
    // Save and clear J, which is used for memory access (TMP1).
    //
    Jr=enable TMP1=load;
    CNSv=0 CNS=enable Jr=load;
    //
    // Restore PC, CS0 and CS1.
    //
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable PCr=load SPi=4;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable CS0=load SPi=4;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
```

```
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable CS1=load SPi=4;
    //
    // Get the original FL, but the FL register is restored at the end
    // of the procedure (TMP0).
    //
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable TMP0=load SPi=4;
    //
    // If the calling process was at supervisor level, the interrupt
    // stack is finished and there is no need to change stack.
    //
    ADDR=iret_transition_back JUMP=conditional JUMP=transition_t;
    //
    // The registers FL and J are restored and the procedure is
    // finished.
    //
    TMP1=enable Jr=load;
    TMP0=enable FL=load_bus             ADDR=fetch JUMP=unconditional;
    //
iret_transition_back:
    //
    // Get DS0, DS1 and SP, but at the moment the registers
    // are not restored (TMP2, TMP3 and TMP4).
    //
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable TMP2=load SPi=4;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable TMP3=load SPi=4;
    DA=sp M=rank32 M=byte3 M=data M=ram_oe;
    DA=sp M=rank32 M=byte2 M=data M=ram_oe;
    DA=sp M=rank32 M=byte1 M=data M=ram_oe;
    DA=sp M=rank32 M=byte0 M=data M=ram_oe M=md_enable TMP4=load SPi=4;
```

```
    //
    // Save SP into SSP
    //
    SPr=enable SSP=load;
    //
    // Restore the loaded register values (included J) and finish.
    //
    TMP2=enable DS0=load;
    TMP3=enable DS1=load;
    TMP4=enable SPr=load;
    TMP1=enable Jr=load;
    TMP0=enable FL=load_bus             ADDR=fetch JUMP=unconditional;
//
in_a:
    // Device access is allowed only to the supervisor, but the device
    // number be read anyway.
    M=rank8 M=byte0 M=code M=ram_oe M=md_enable IOA=load PCi=1
                        ADDR=fetch JUMP=conditional JUMP=supervisor_f;
    IOC=req;
    CTRL=nop;
    IOC=enable Ar=load FL=update_bus    ADDR=fetch JUMP=unconditional;
in_b:
    // Device access is allowed only to the supervisor, but the device
    // number be read anyway.
    M=rank8 M=byte0 M=code M=ram_oe M=md_enable IOA=load PCi=1
                        ADDR=fetch JUMP=conditional JUMP=supervisor_f;
    IOC=req;
    CTRL=nop;
    IOC=enable Br=load FL=update_bus    ADDR=fetch JUMP=unconditional;
out_a:
    // Device access is allowed only to the supervisor, but the device
    // number be read anyway.
    M=rank8 M=byte0 M=code M=ram_oe M=md_enable IOA=load PCi=1
                        ADDR=fetch JUMP=conditional JUMP=supervisor_f;
    Ar=enable IOC=load;
    IOC=req                             ADDR=fetch JUMP=unconditional;
out_b:
    // Device access is allowed only to the supervisor, but the device
    // number be read anyway.
    M=rank8 M=byte0 M=code M=ram_oe M=md_enable IOA=load PCi=1
```

```
                        ADDR=fetch JUMP=conditional JUMP=supervisor_f;
    Br=enable IOC=load;
    IOC=req                              ADDR=fetch JUMP=unconditional;
memcpy:
    //
    // Save the index registers used.
    //
    SPr=enable TMP0=load;
    Ir=enable TMP1=load;
    Jr=enable TMP2=load;
    //
    // Load arguments: source --> I, destination --> SP, size --> J
    //
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable Ir=load PCi=4;
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable SPr=load PCi=4;
    M=rank32 M=byte3 M=code M=ram_oe;
    M=rank32 M=byte2 M=code M=ram_oe;
    M=rank32 M=byte1 M=code M=ram_oe;
    M=rank32 M=byte0 M=code M=ram_oe M=md_enable Jr=load FL=update_bus
        PCi=4;
memcpy_check:
    //
    // If the size is zero, the cycle is finished.
    //
    ADDR=memcpy_restore JUMP=conditional JUMP=zero_t;
    Ji=-1;
    DA=i M=rank8 M=byte0 M=data M=ram_oe M=md_enable TMP3=load;
    TMP3=enable DA=sp M=rank8 M=byte0 M=data M=ram_we;
    Jr=enable FL=update_bus ADDR=memcpy_check JUMP=unconditional;
memcpy_restore:
    TMP0=enable SPr=load;
    TMP1=enable Ir=load;
    TMP2=enable Jr=load                  ADDR=fetch JUMP=unconditional;
    //
```

```
st_a_ssp:
    ADDR=fetch JUMP=conditional JUMP=supervisor_f;
    Ar=enable  SSP=load                 ADDR=fetch JUMP=unconditional;
halt:
    CTRL=stop;
//
end
```

13.23 Process mode and interrupts

«

The main difference from any previous version of this experimental CPU lies in the distinction between two working modes: supervisor and user modes. When the CPU is started, the ***FL*** register is reset to 80_{16}, so that the supervisor flag is set. When the supervisor flag is set, all other flags can be changed, but when it is clear the flags $\boldsymbol{FL}_{7:4}$ cannot be changed by the process (except the IRQ request that is independent from the process).

Supervisor processes can be interrupted (from the hardware or by software), when it happens the process is suspended and some data is pushed onto the stack: ***FL***, ***CS1***, ***CS0*** and ***PC***. Then the code segment register are cleared, because the interrupt code must be located on an absolute memory address. The data segment is not changed and the process stack is used. During the interrupt process, the IRQ enable flag is cleared.

When a user process is interrupted, the registers that should be saved into the stack are saved in some other temporary registers, then the stack pointer is replaced with the value contained inside ***SSP*** and then the old values are pushed: ***SP***, ***DS1***, ***DS0***, ***FL***, ***CS1***, ***CS0***, ***PC***. During the interrupt process, the IRQ enable flag is cleared, the supervisor flag is set and the transition flag is set.

When returning from an interrupt, the transition flag is checked: if it is set, the stack is restored into temporary registers and then the register ***SSP*** is used to save the final stack pointer location. Then the user process registers previously saved are restored and it continues to run with the previous privileges.

The preferred way to create a subprocess is to pretend that the process was interrupted, forging the stack that the interrupt would have produced, using then the **`iret`** instruction. The last example shown in the following sections is about the creation of user mode subprocesses.

13.24 Macrocode example: 'memcpy'

The following example shows the use of the instruction **memcpy**; it is requested to copy 32 bytes from the label ***data_0*** to the label ***data_1***.

Listing 13.62. Macrocode example using **memcpy**.

```
begin macrocode @ 0
start:
    memcpy #data_0, #data_1, 32
    halt
data_0:
    int32 0x12345678
    int32 0x9ABCDEF0
    int32 0x12345678
    int32 0x9ABCDEF0
    int32 0x12345678
    int32 0x9ABCDEF0
    int32 0x12345678
    int32 0x9ABCDEF0
data_1:
    int32 0x00000000
    int32 0x00000000
    int32 0x00000000
    int32 0x00000000
    int32 0x00000000
    int32 0x00000000
    int32 0x00000000
    int32 0x00000000
end
```

Figure 13.63. RAM memory content, before and after execution.

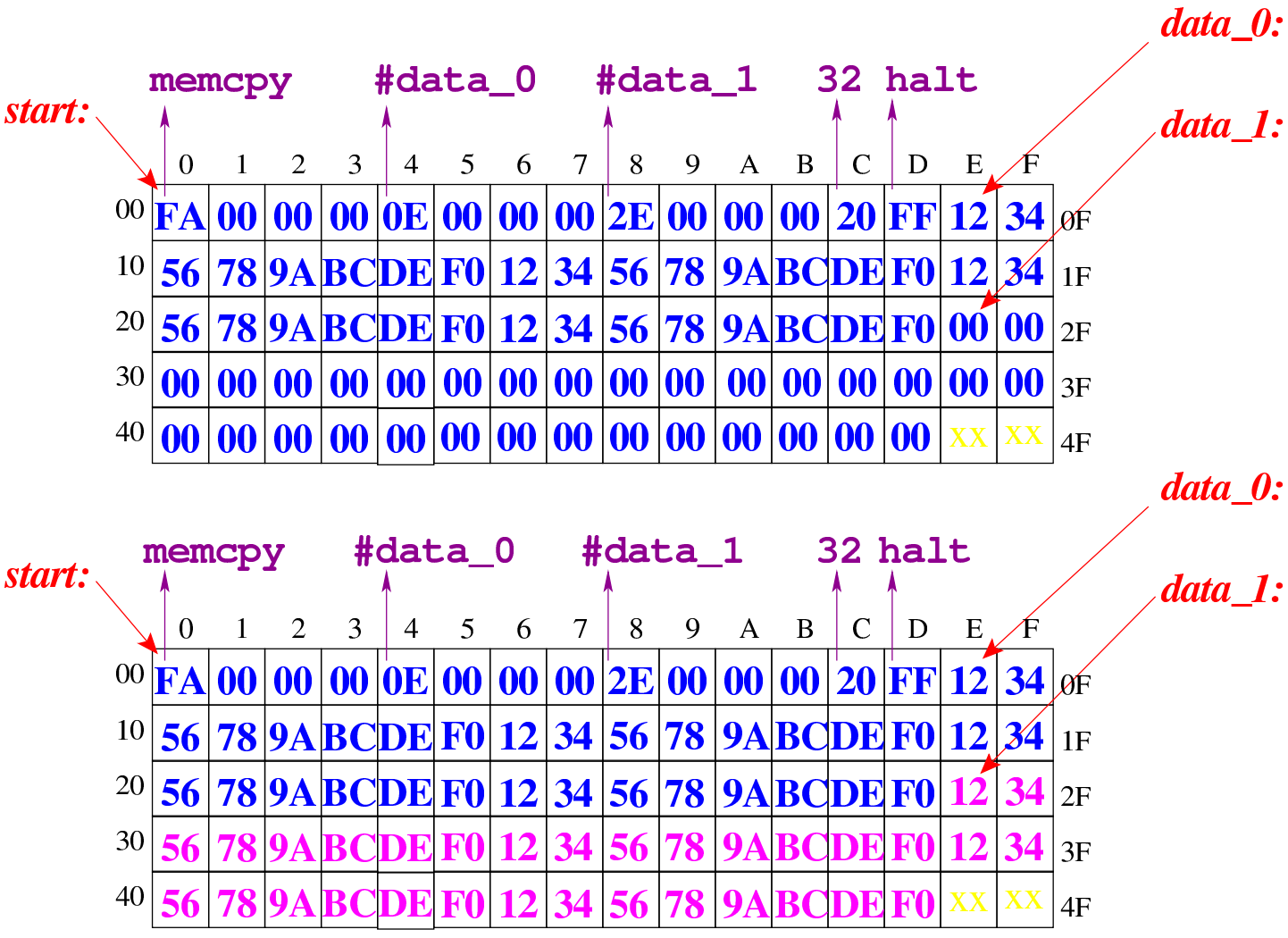
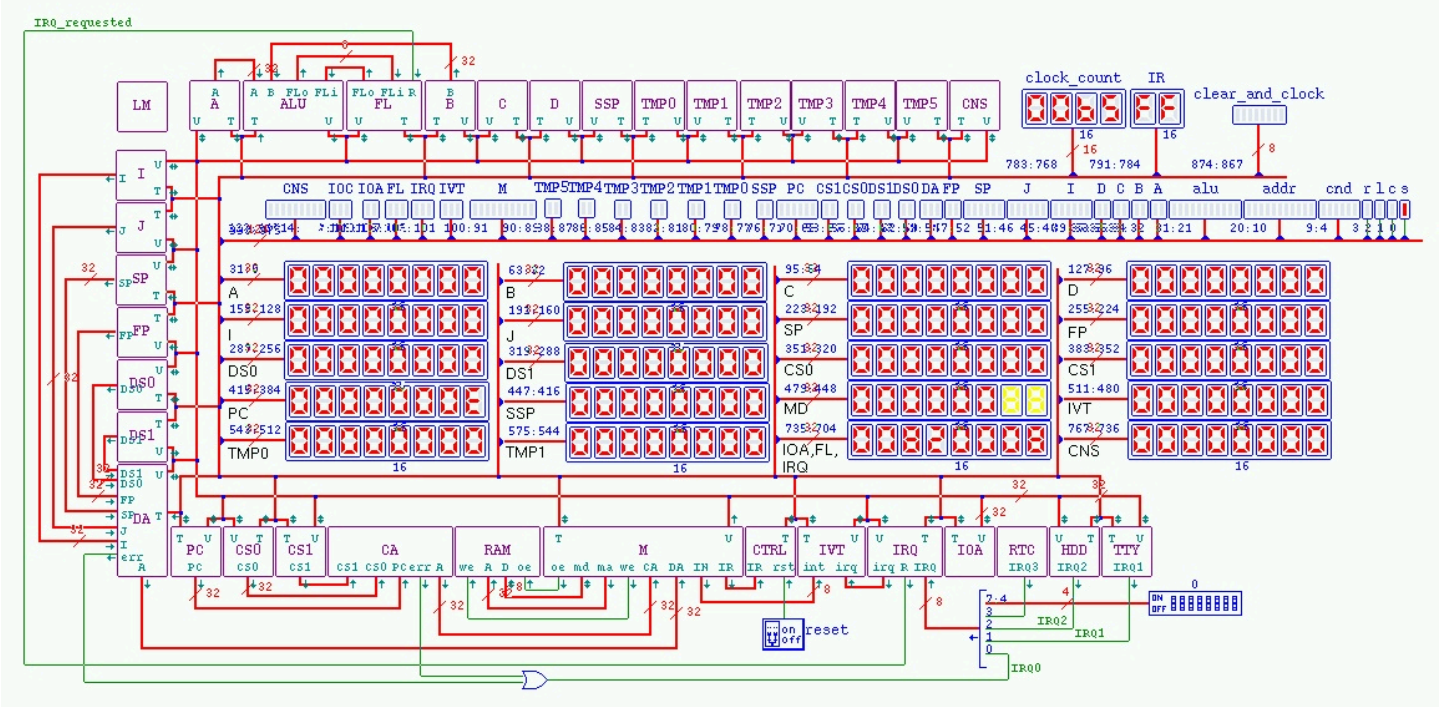


Figure 13.64. End of execution of the example above. Video: ogv *http://www.youtube.com/watch?v=PfFRAaZXWOQ*



13.25 Macrocode example: TTY with interrupt

The following example shows the IVT setup, IRQ activation and TTY echo (the typed text is echoed on the virtual screen): each keypress starts a IRQ1 signal that is processed without priority change, because everything runs as supervisor.

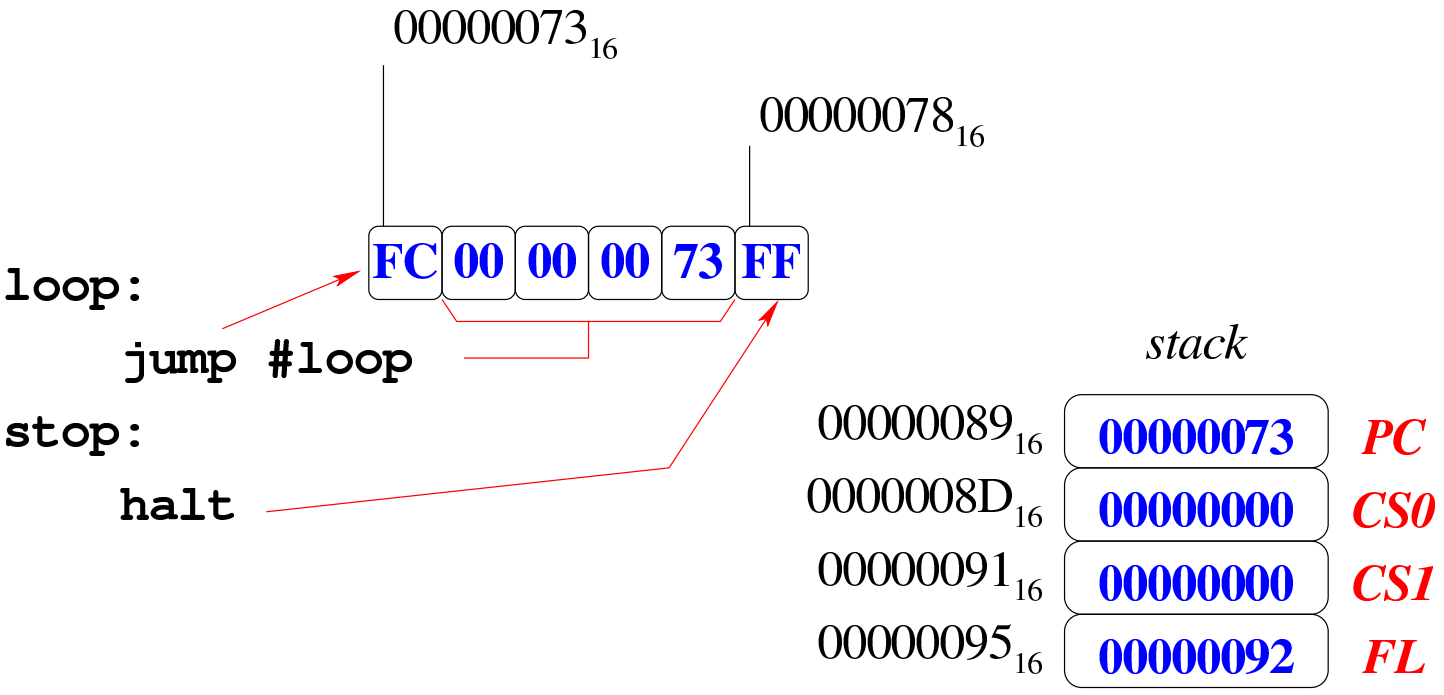
Listing 13.65. Macrocode example.

```
begin macrocode @ 0
    jump #start
    nop
    nop
    nop
interrupt_vector_table:
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
```

```
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // IRQ segmentation fault
    int32 #keyboard                    // IRQ keyboard
    int32 #default_interrupt_routine   // IRQ hard disk
    int32 #default_interrupt_routine   // IRQ real time clock
    int32 #default_interrupt_routine   // IRQ
    int32 #default_interrupt_routine   // IRQ
    int32 #default_interrupt_routine   // IRQ
    int32 #default_interrupt_routine   // IRQ
default_interrupt_routine:
    iret
keyboard:
    in %A,1                            // keyboard read
    jump %zero,1,#keyboard_end         // if zero exit
    out %A,0                           // print on screen
    jump #keyboard                     // continue
keyboard_end:
    iret
keyboard_reset:
    in %A,1                            // keyboard read.
    jump %zero,0,#keyboard_reset       // if not zero continue
    return
start:
    //
    // Set initial SP value.
    //
    load32 0, #stack_bottom
    store %A, %SP
    //
    // Reset the keyboard.
    //
    call #keyboard_reset
    //
    // Set up the IVT
    //
    ivtl #interrupt_vector_table
```

```
    //
    imrl 0xF7       // Allow all IRQs, except RTC.
    flag_i 1        // IRQ enabled.
    //
loop:
    //
    // Just wait for interrupt.
    //
    jump #loop
stop:
    halt
    //
stack_top:
    int32 0xFFFFFFFF
    int32 0xFFFFFFFF
    int32 0xFFFFFFFF
    int32 0xFFFFFFFF
    int32 0xFFFFFFFF
    int32 0xFFFFFFFF
    int32 0xFFFFFFFF
    int32 0xFFFFFFFF
stack_bottom:
end
```

Figure 13.66. Stack after the IRQ1 begins to be processed.



Video: ogv *http://www.youtube.com/watch?v=JK0RUuKTwZQ*

13.26 Macrocode example: user and supervisor

«

This version of the experimental CPU should be able to distinguish from supervisor processes with privileges and user process without privileges. The following code is a simple program playing with the stack, replacing values inside the registers ***A***, ***B*** and ***C***. A the moment, this process runs with the default supervisor privileges.

Listing 13.67. A simple process.

```
begin macrocode @ 0
start:
    load32 0, 0x12345678
    push %A
    push %A
    call #subroutine
    pop %B
    pop %C
    jump #start
subroutine:
    load32 0, 0x9ABCDEF0
```

```
    push %A
    push %A
    pop %B
    pop %C
    return
stop:
    halt
stack_top:
        int32 0xFFFFFFFF
        int32 0xFFFFFFFF
        int32 0xFFFFFFFF
        int32 0xFFFFFFFF
        int32 0xFFFFFFFF
        int32 0xFFFFFFFF
        int32 0xFFFFFFFF
        int32 0xFFFFFFFF
        int32 0xFFFFFFFF
        int32 0xFFFFFFFF
stack_bottom:
end
```

This program saves some space for the stack, but the ***SP*** register is not initialized and in fact it start from the real bottom of the memory. Video: ogv *http://www.youtube.com/watch?v=Vf1vDOztE1I* .

Apart the fact that at the moment the stack is placed at the very bottom of the memory, as it is declared on the above script it occupies 70 bytes (46_{16}). The following program should run in supervisor mode and it should let start the user process (in user mode). The source listed below does not contain the code for the user process, instead it declares a space at the bottom, from ***user_process_start*** and ***user_process_end***; that space is of the same size of the user process: 70 bytes.

Listing 13.68. A supervisor process with space for a user process.

```
begin macrocode @ 0
    jump #start
    nop
    nop
    nop
interrupt_vector_table:
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // unassigned
    int32 #default_interrupt_routine   // IRQ segmentation fault
    int32 #keyboard                    // IRQ keyboard
    int32 #default_interrupt_routine   // IRQ hard disk
    int32 #default_interrupt_routine   // IRQ real time clock
    int32 #default_interrupt_routine   // IRQ
    int32 #default_interrupt_routine   // IRQ
    int32 #default_interrupt_routine   // IRQ
    int32 #default_interrupt_routine   // IRQ
default_interrupt_routine:
    iret
op_code_error:
    halt
keyboard:
    in %A,1                            // keyboard read
    jump %zero,1,#keyboard_end         // if zero exit
    out %A,0                           // print on screen
    jump #keyboard                     // continue
keyboard_end:
    iret
keyboard_reset:
    in %A,1                            // keyboard read.
```

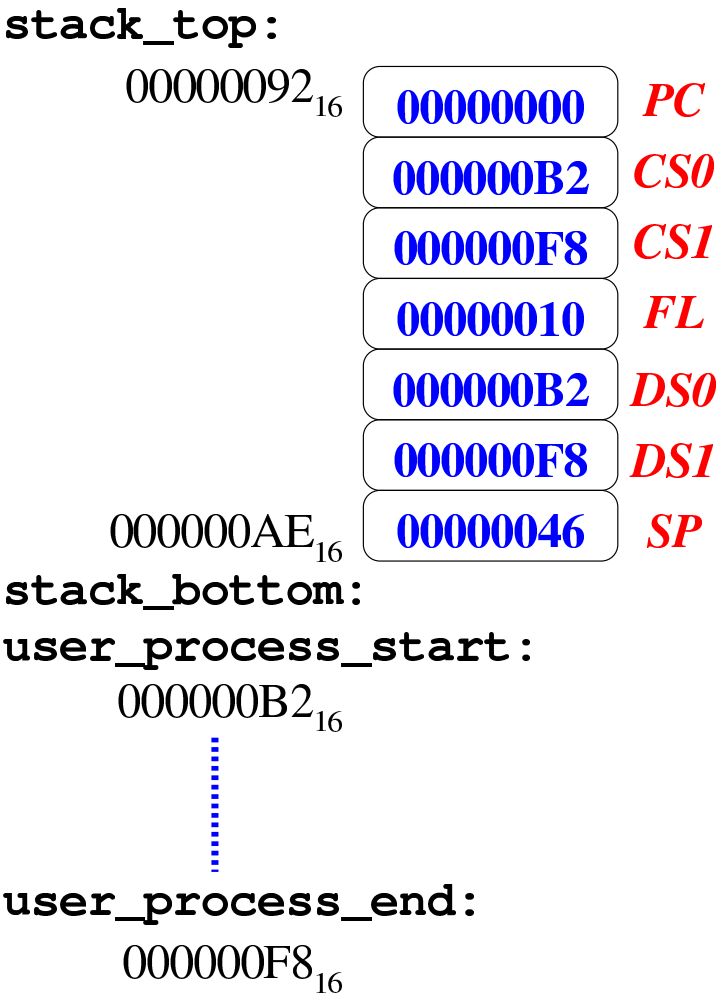
```
    jump %zero,0,#keyboard_reset    // if not zero continue
    return
start:
    //
    // Set initial SP value.
    //
    load32 0, #stack_bottom
    store %A, %SP
    //
    // Reset the keyboard.
    //
    call #keyboard_reset
    //
    // Set up the IVT
    //
    ivtl #interrupt_vector_table
    //
    flag_i 0       // IRQ disabled.
    imrl 0xF7      // Allow all IRQs, except RTC.
    //
    // Build the user process stack.
    //
build_interrupted_stack:
    load32 0, 0x00000046          // SP
    push %A
    //
    load8 0, #user_process_end    // DS1
    push %A
    load8 0, #user_process_start // DS0
    push %A
    //
    load8 0, 0x10         // IRQ enabled
    push %A
    //
    load8 0, #user_process_end    // CS1
    push %A
```

```
    load8 0, #user_process_start // CS0
    push %A
    //
    load32 0, 0x00000000         // PC
    push %A
    //
    flag_t 1
run_user:
    iret
    //
stop:
    halt
stack_top:
        int32 0xCCCCCCCC
        int32 0xCCCCCCCC
        int32 0xCCCCCCCC
        int32 0xCCCCCCCC
        int32 0xCCCCCCCC
        int32 0xCCCCCCCC
        int32 0xCCCCCCCC
        int32 0xCCCCCCCC
stack_bottom:
user_process_start:
        int16 0x1111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
```

```
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
        int32 0x11111111
user_process_end:
end
```

Please notice that starting at the label ***build_interrupted_stack*** the user return stack is created and the flags are modified asserting that there 'was' a transition (an interrupt from a user process): that is why the stack is builded including also the registers ***SP***, ***DS1*** and ***DS0***. Please notice also that the value pushed into the stack for the ***FL*** register would enable IRQs when the user process will be running. The following figure shows how should appear the process stack builded by the supervisor process.

Figure 13.69. Interrupted process stack, before the user process is effectively run.



The program counter (***PC***) of the user process is saved with the value zero, because it is relative to the memory segment for data and for code (***DS0*** and ***CS0***).

To build the memory image of the supervisor process and the user process, some more work is necessary. TKGate produces the following memory code for the user process, as it was builded at the beginning of this example section:

```
...
@0
46 12 34 56 78 70 70 e6
0 0 0 13 79 7a fc 0
0 0 0 46 19a bc de f0
70 70 79 7a e7 ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff
```

On the other hand, the program representing the supervisor process produces the following memory image:

```
...
@0
fc 0 0 0 61 0 0 0
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 4a
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
f5 ff f6 1 eb 0 0 0
58 f8 0 fc 0 0 0 4a
f5 f6 1 ea 0 0 0 59
```

```
e7 46 0 0 0 b2 6 e6
0 0 0 59 f3 0 0 0
8 e0 f2 f7 46 0 0 0
46 70 40 f8 70 40 b2 70
40 10 70 40 f8 70 40 b2
70 46 0 0 0 0 70 e3
f5 ff cc cc cc cc cc cc
cc cc cc cc cc cc cc cc
cc cc cc cc cc cc cc cc
cc cc cc cc cc cc cc cc
cc cc 11 11 11 11 11 11
11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11
```

The user process memory image should be merged into the supervisor one, with the help of a text editor, like this:

```
...
@0
fc 0 0 0 61 0 0 0
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 4a
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
0 0 0 48 0 0 0 48
f5 ff f6 1 eb 0 0 0
```

```
58 f8 0 fc 0 0 0 4a
f5 f6 1 ea 0 0 0 59
e7 46 0 0 0 b2 6 e6
0 0 0 59 f3 0 0 0
8 e0 f2 f7 46 0 0 0
46 70 40 f8 70 40 b2 70
40 10 70 40 f8 70 40 b2
70 46 0 0 0 0 70 e3
f5 ff cc cc cc cc cc cc
cc cc cc cc cc cc cc cc
cc cc cc cc cc cc cc cc
cc cc cc cc cc cc cc cc
cc cc 46 12 34 56 78 70
70 e6 0 0 0 13 79 7a
fc 0 0 0 0 46 19a bc
de f0 70 70 79 7a e7 ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff
```

Video: ogv *http://www.youtube.com/watch?v=ssFuzyj1wo8* .

The above video shows what the whole thing does: the supervisor creates the return stack for user process and then it (the supervisor) pretends to return from an interrupt signal that was never fired. The user process start living with limited privileges an it does what it does all over again. When a key is pressed on the terminal, an IRQ is fired and the user process is interrupted. The interruption implies a privilege change that requires transition to the supervisor stack. After the interrupt is processed (the key is shown on the terminal screen), if no other pressed keys are waiting, the user process is restored returning to the previous user stack.

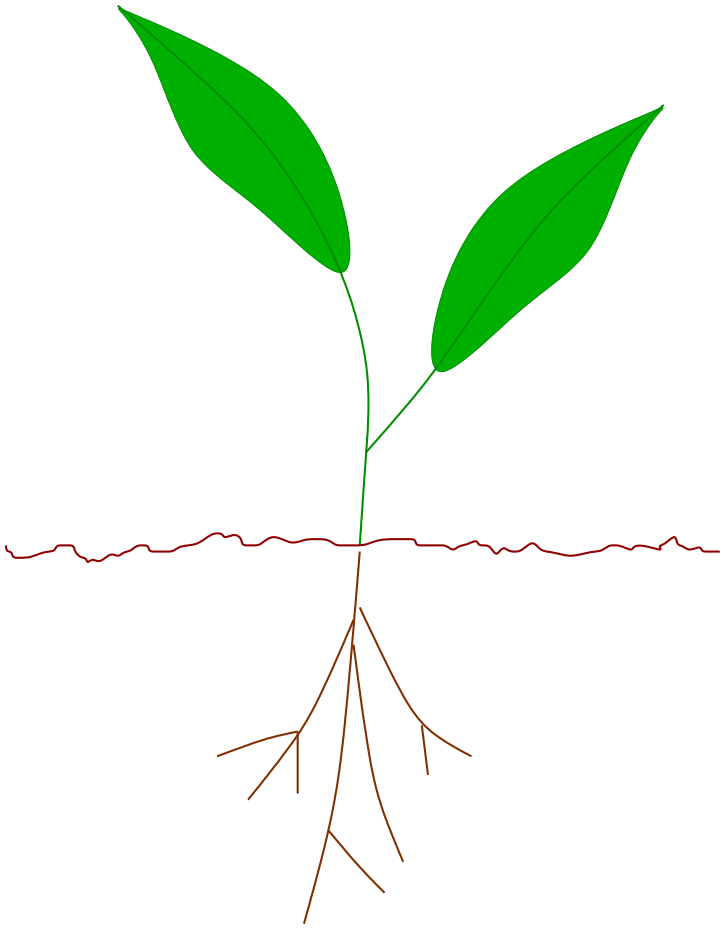
References

- *TKGate*, *http://www.tkgate.org*
- Albert Paul Malvino, Jerard A. Brown, *Digital Computer Electronics*, Glencoe/Mcgraw-Hill *http://www.amazon.com/Digital-Computer-Electronics-Albert-Malvino/dp/0028005945*
- Morris M. Mano, *Computer system architecture*, Prentice-Hall *http://www.amazon.com/Computer-System-Architecture-Morris-Mano/dp/0131755633*
- Andrew S. Tanenbaum, *Structured computer organization*, Prentice-Hall *http://www.amazon.com/Structured-Computer-Organization-Andrew-Tanenbaum/dp/0131485210*
- John P. Hayes, *Computer architecture and organization*, McGraw-Hill *http://www.amazon.com/Computer-Architecture-Organization-John-Hayes/dp/0070273553*
- Olivier Carton, *Circuits et architecture des ordinateurs*, *http://www.liafa.jussieu.fr/~carton/Enseignement/Architecture/archi.pdf*

Index

51 ! 54 != 54 !== 54 * 54 */ 33 + 54 . 33 / 54 /* 33 // 33 2's complement 132 74138 244 74139 247 74148 271 74150 257 74151 262 74153 265 74154 250 74157 268 74181 281 287 74182 291 74238 254 74381 293 74382 297 7483 275 7485 278 ; 33 == 54 === 54 ?: 54 active high 10 active low 10 adder 114 addition 156 addressing 58 ALU 177 179 always 68 AND 8 and 52 arithmetic shift 107 array 43 58 assert 10 assign 62 assignment 69 begin 68 behavioral module 68 binary encoder 99 bit-size 152 bit range 58 bit size 43 58 bit slice 179 blocking assignment 69 boolean algebra 13 bounce free 193 buf 52 buffer 7 bufif0 52 bufif1 52 bus 229 carry 150 carry lookahead 125 case 78 casex 78 casez 78 combinational circuit 62 83 combinational network 83 comparation 162 control structure 78 counter 219 220 counter register 217 data distributor 90 data register 212 data selector 93 data type 43 decoder 87 defparam 51 delay 71 185 demultiplexer 90 demux 90 De Morgan's laws 13 directive 38 discrete logic 25 division 171 D flip-flop 199 Earle latch 199 edge-triggered 195 encoder 99 end 68 endmodule 48 event 43 76 event expression 74 expression 54 fast adder 125 fetch 309 flag 150 flip-flop 185 195 floating 25 floating point number 41 for 78 forever 78 fork 81 full-adder 114 function 60 fundamental product 17 gated D latch 199 gate instance 52 half-adder 114 half-subtractor 119 hardware description language 28 HDL 28 31 high impedance 25 hold time 198 IC 25 if 78 initial 68 inout 46 input 46 integer 43 integer number 41 integrated circuit 25 inverter 7 JK flip-flop 204 207 join 81 Karnaugh map 19 latch 185 level event 72 literal constant 41 logic circuit 10 logic connective 5 logic gate 5 logic network 10 logic operator 5 logic shift 107 logic unit 106 logic variable 5 macro 38 macrocode 309 master-slave 199 memory 43 185 microcode 236 309 microcode word 309 microinstruction 309 module 48 module instance 51 multiplexer 93 multiplication 165 nand 52 negate 10 negative 150 net 43 netlist module 62 non-blocking assignment 69 nor 52 not 52 NOT 7 notif0 52 notif1 52 not-inverter 7 nxor 52 opcode 309 operation code 309 OR 9 or 52 output 46 overflow 150 156 158 parameter 48 port 46 primitive 52 primitive instance 52 priority encoder 102 procedural assignment 69 propagation delay 25 185 rank 150 152 real 43 recovery time 198 reg 43 register 211 removal time 198 repeat 78 ripple-carry 114 rotation 107 158 scalar 43 sequential circuit 185 setup time 198 shift 107 158 shift register 216 sign 150 154 signal 10 41 size 152 SR flip-flop 195 SR latch 188 status 150 string 41 59 subtraction 119 156 sum of products 17 supply0 43 supply1 43 switch 193

switch instance 52 symbolic macro 38 system function 76 system task 76 thread 68 thread 81 time 43 top level module 48 transparent latch 199 tri 43 tri0 43 tri1 43 triand 43 trior 43 tri-state buffer 25 229 two's complement 132 T flip-flop 203 variable 43 vector 43 58 Verilog 28 31 wait () 72 wand 43 while 78 wire 43 wor 43 x 41 XOR 9 xor 52 z 41 Z 25 zero 150 & 54 && 54 @ 74 ^ 54 ~ 54 ~& 54 ~^ 54 ~| 54 | 54 || 54 {{}} 54 {,} 54 $display() 33 76 $finish 76 $finish() 33 $monitor() 76 $time 76 – 54 < 54 <= 54 << 54 <<< 54 > 54 >= 54 >> 54 >>> 54 % 54 `define 38 `else 38 `endif 38 `ifdef 38 `ifndef 38 `timescale 38



“from the ground up” electronic edition, ISBN 0000000000